NTNU
Norwegian University of
Science and Technology

A Simulator for the Development of Autonomous Robots

Christer Mathiesen

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem statement

An autonomous robot needs to be able to locate itself in its environment and make informed decisions based on the surroundings. In addition to measuring states pertaining to the state or the robot, like inertial measurements, autonomy requires sensory interaction with the environment to gather information. Typically this can include recording camera images and track features from frame to frame to estimate motion relative to the scene, and active laser ranging to measure distances as a complementary and redundant way to understand scene structure.

Inherently, an autonomous robot is an embedded computer system running algorithms to perform *navigation* based on sensor fusion to estimate location, *guidance* to plan movement trajectories and *control* to execute them. This poses a considerable challenge during systems development, as the software is not testable in and of itself without being embodied in the robot hardware interacting with an external environment. It might be impractical, time-consuming and even dangerous to include all or a part of the physical components of an embedded system in day to day development. Therefore it is often beneficial to emulate some parts of the system hardware and software in a simulation.

To aid the development of a set of navigation and control algorithms for an autonomous robot we wish to replace all input/output capabilities and interaction with the physical world with a standalone simulator interfacing with the rest of the system completely *as is*. In terms of its inputs and outputs, the simulator must provide

1. Force and torque actuation on a quadrotor body in response to control **input**.
2. Six degrees of freedom rigid body dynamics resulting from the actuation.
3. Sensor data **output** for the navigation filter to provide control feedback, including
   - inertial measurements,
   - camera images,
   - laser scanner range measurements.

Preface

This thesis is submitted to the Department of Engineering Cybernetics at the Norwegian University of Science and Technology, as the final compulsory part of the degree of Master of Science in Engineering Cybernetics. It is directly motivated by, and done in the context of, the startup company *Versor* with work starting late fall 2016. In that effect, the work has not been done in isolation, but I acknowledge that the work presented here is my own. While the added overhead of starting a business when you are supposed to be writing your thesis is demanding, it is an obvious reward that the results of this work as it stands not only will be the culmination of far too many years of school, but serve an active role in ongoing commercial software development.

To my supervisor, Tor Onshus, I want to display gratitude for accepting my thesis proposal and being patient with the presentation of evidence, or lack thereof, that progress was being made on the thesis. I want to show appreciation to my Versor co-founders, Eirik Worren Legernæs and Erlend Sierra. Without you I would not have anything better to do with my time than to actually write my thesis. I am looking forward to continue working with you making a convincing showing that we seemingly know what we are doing. So far it has been an experience I would never be without. My only regret is that we didn't get an even bigger camera calibration board for our office. Finally, I want to thank my parents, who I know will be reading this, for always supporting and believing in me, especially when it was needed the most. You are great, and I love you.

Christer Mathiesen
Trondheim, June 5, 2017

Abstract

A simulator for the development of vision-based navigation, guidance and control algorithms of an autonomous flying robot is implemented. The simulator is a drop-in replacement for all system input/output and interfaces with the rest of the system *as is*. Synthetic data is generated in the form of inertial measurements, camera images and scanning laser range measurements. Computer graphics are used to generate image projections of a virtual scene, emulating calibrated intrinsic parameters of a specific camera. The effects of optical distortion from a wide angle lens are emulated using iterative methods, and the apparent duality between image distortion and undistortion methods is presented. Depth images are produced to calculate distance measurement to specific points in the virtual scene, and a depth buffer sampling model is created to generate point cloud scans for a simulated scanning laser sensor. The sensor data is used as input to a sensor fusion filter providing navigation feedback to guidance and control algorithms running in real-time. General six-degrees of freedom rigid body dynamics is simulated with a quadrotor actuation model responding to control input, closing the loop between control and navigation allowing for full system integration and testing on the desktop with minimal overhead. Visualization tools are demonstrated as a graphical interface to interact with the headless simulation, for real-time visual verification of system status.

Sammendrag

En simulator er implementert for å forenkle utviklingen av kamera-basert navigasjon, guidance og kontroll av en autonom flyvende robot. Simulatoren erstatter all system-input/output og kommuniserer med resten av systemet *som det er*. Syntetiske data genereres i form av treghetsmålinger, kamerabilder og avstandsmålinger. Datagrafikk brukes for å generere spesifikke kamerabilder av en virtuell scene. Forvrengningseffekten av en bredvinklet linse er emulert gjennom iterative metoder, og det presenteres en åpenbar dualitet mellom forvrengning og retting av bilder. Dybdebilder produseres for å beregne avstanden til spesifikke punkter i scenen, og en modell for sampling av dybdebufferet framstilles for å generere en punktsky for en simulert laserskanner. Sensordataen benyttes i et navigasjonsfilter for kontroll-tilbakekobling i ekte tid. Dynamikk i seks frihetsgrader for generelle stive legemer med en modell av et kvadkopter simuleres i henhold til kontrollinput, for lukket sløyfe verifisering og full systemintegrasjonstesting. Et grafisk grensesnitt demonstreres for å visualisere systemstatus i real-time.

Contents

List of Figures **xiv**

List of Tables **xv**

1 Introduction **1**

1.1 Background . . . 1
1.2 Motivation . . . 2
1.3 Objectives . . . 4
1.4 Contributions . . . 5
1.5 Outline . . . 5

2 Rigid Body Dynamics **7**

2.1 Kinematics . . . 7
2.1.1 Coordinate frames . . . 7
2.1.2 Frames of reference . . . 8
2.1.3 Differential equations . . . 9
2.2 Kinetics . . . 10
2.2.1 Inertial frames of reference . . . 10
2.2.2 Newton-Euler equation of motion . . . 11
2.3 Navigational kinematics . . . 13
2.3.1 Inertial reference frame approximations . . . 13
2.3.2 Earth-centered, Earth-fixed reference frame . . . 14
2.3.3 Local-level navigational reference frame . . . 17
2.4 Numerical solution . . . 18
2.4.1 Angular integration . . . 18
2.4.2 Linear integration . . . 19

3 Quadrotor Actuation **21**

3.1 Forces and torques . . . 21
3.2 Quadrotor model . . . 22
3.3 Rotor model . . . 23
3.3.1 Propeller aerodynamics . . . 23
3.3.2 DC motor electromechanics . . . 24
3.3.3 Electronic speed controller dynamics . . . 25

4 Inertial Measurements **29**
4.1 Accelerometer . . . 29
4.1.1 Measurement model . . . 30
4.1.2 Generating a measurement . . . 30
4.2 Gyroscope . . . 31
4.2.1 Measurement model . . . 31
4.2.2 Generating a measurement . . . 32

5 Camera Images **33**
5.1 Pinhole camera . . . 33
5.1.1 Pinhole camera model . . . 33
5.1.2 Pixel coordinate transform . . . 34
5.2 OpenGL camera . . . 35
5.2.1 Viewing Frustum . . . 35
5.2.2 Projection matrix . . . 36
5.2.3 View matrix . . . 37
5.3 Constructing OpenGL camera from pinhole camera . . . 38
5.3.1 Projection matrix . . . 38
5.3.2 View matrix . . . 39
5.4 Optical distortion . . . 40
5.4.1 Distortion models . . . 40
5.4.2 Fisheye camera model . . . 41
5.4.3 Image distortion . . . 41
5.4.4 Image undistortion . . . 44

6 Laser Range Measurements **47**
6.1 Distance measurement . . . 47
6.2 2D laser scan . . . 49
6.2.1 Frustum subdivision . . . 50
6.2.2 Sampling . . . 52
6.2.3 View matrix . . . 53
6.3 3D laser scan . . . 53
6.4 Point cloud . . . 54

7 Implementation **55**
7.1 Rigid body dynamics . . . 57
7.2 Quadrotor actuation . . . 57
7.3 Inertial measurements . . . 59
7.4 Camera images . . . 59
7.5 Laser range measurements . . . 61

8 Results **65**
8.1 Vehicle dynamics . . . 65
8.2 Camera images . . . 67
8.2.1 Validation by navigation . . . 69
8.3 Laser range measurements . . . 71
8.3.1 Occupancy map . . . 72

8.4 Final remarks 73

9 Conclusion **75**

9.1 Further work 76

References **77**

List of Figures

1.1 Simplified autopilot architecture 3

2.1 Vector $\vec{x}$, and unit axes of coordinate frames a and b 7
2.2 Configuration of two arbitrarily moving reference frames 8
2.3 Global ECEF frame, local navigational frame, body frame 15
2.4 Reference ellipsoid and normal gravity model 16
2.5 Integration of angular velocity 18
2.6 Integration of velocity 19

3.1 Quadrotor lift forces and resulting moments about body center 22
3.2 Torques generated by the four spinning rotor actuators 22
3.3 Speed feedback control of brush-less DC motor 26

5.1 Pinhole camera projection 34
5.2 Pinhole camera projection to pixel coordinates 35
5.3 Viewing frustum 36
5.4 OpenGL perspective frustum and camera frame 37
5.5 Perspective unprojection 38
5.6 Direct method of image distortion 41
5.7 Indirect method of image distortion 43
5.8 Direct method of image undistortion 44
5.9 Indirect method of image undistortion 45
5.10 Duality of image distortion and undistortion 46

6.1 The scene depth of a point 48
6.2 2D laser scanner sampling points in the plane 49
6.3 Lidar frustum subdivision 50
6.4 Sampling points at equidistant angles in one frustum scan 51
6.5 Sampling points in the vertical field of view 53

7.1 Interaction between headless simulator, flight code and visualization . . . 55
7.2 Rendering of the simulated quadrotor body 56
7.3 Transient rotor output in response to an increasing input 58
7.4 Image taken with the IDS camera 60
7.5 Direct image distortion and undistortion 63

7.6 Indirect image distortion and undistortion 64

8.1 Real-time rendering of the quadrotor flying in a virtual scene. 65
8.2 Navigational estimate of the body frame pose 66
8.3 Simulated camera images with fish-eye distortion 68
8.4 Trajectory of estimated positions from simulated run 69
8.5 Estimated position and orientation from simulated run 70
8.6 Visualization of lidar point cloud 71
8.7 Occupancy map generated with lidar measurements 72
8.8 Side-by-side comparison of a simulated run versus a real run 74

List of Tables

7.1 Defining parameters of the WGS84 reference ellipsoid 57
7.2 Quadrotor rigid body parameters 57
7.3 Rotor model parameters 58
7.4 Accelerometer and gyroscope parameters 59
7.5 Camera projection parameters 60
7.6 Camera distortion parameters 61
7.7 Lidar parameters 62
7.8 Lidar model parameters 62

1 Introduction

This thesis will document the principal implementation of a simulator to aid the development of navigation, guidance and control algorithms for an autonomous flying robot. Specifically, the work will focus on generating synthetic sensor data and simulating appropriate vehicle dynamics in response to control inputs.

1.1 Background

Automation of mobile robots presents a significant opportunity for industry to save time and cost and reduce the need for human involvement in potentially dangerous work. Mobile robots with a sufficient degree of autonomy can operate with little to no supervision where humans cannot or should not work. We will consider the defining characteristic of an autonomous robot regardless of high-level goal as being able to locate itself within its environment and make informed decisions on where to go and what to do based on the surroundings. It is not enough for the robot to be concerned with its own state and measure signals pertaining to the system only, like inertial measurements and satellite navigation signals, but to make use of *exteroceptive* sensors to acquire information about the environment. Typically this can include recording camera images and track features from frame to frame to estimate motion relative to the surroundings, and laser ranging to measure distances as a complementary and redundant way to understand scene structure.

An autonomous robot is inherently an embedded computer system running algorithms to perform *navigation* to fuse sensor data and estimate location, *guidance* to initiate locomotion based the location and an understanding of the environment, and *control* to actuate those decisions into movement. This poses a considerable challenge during system development, as the software is not testable in and of itself without being embodied in the robot hardware interacting with an external environment. It might be impractical, time-consuming and even dangerous to include all or even a part of the physical components of an embedded system in day to day development. Therefore it is often beneficial to emulate in a simulation some parts of the system hardware and software.

Simulation is widely used when it comes to the *design* of embedded control systems. Typically a mathematical model of the vehicle or plant is developed and run as a simulation in a software prototyping framework, and the desired control and navigation algorithms are designed in the same prototyping environment and tested via the simulated model, emulating the performance of the final embedded system interacting with the physical world. However, the simulation tools used during the initial design phase can be extended and used favourably during the development of the embedded software where the actual embedded algorithms interact with a fully or partially simulated external system. This is called *hybrid simulation* or *hardware-in-the-loop* simulation[5].

In its penultimate form, hardware-in-the loop simulation involves the embedded system perform completely *as is* during development and testing, including sensors and actuators. Only the process itself is simulated, and its dynamic response to actuators and the resulting sensor input. This necessitates the simulation itself being embedded in hardware, to perform A/D conversions and possibly mechanically actuate to stimulate the appropriate signals in the sensor measurements. Obviously artificially generating physical sensor inputs may require considerable realization efforts[13]. This level of emulation is often impractical or downright impossible depending on the sensor in question, but nonetheless offers the most real testing conditions possible for executing an embedded system and has been employed with success to assess the performance of satellite launcher control systems, real-time flight simulation training and in aerospace and automotive design.

Differentiating between the *control system* being tested, executing the *real* algorithms on *real* hardware, and the *controlled system* including process dynamics, sensor and actuator hardware, a considerably more affordable approach is to simulate the controlled system in full. This still involves A/D conversions to interact with the control system over the same electrical interface, but is considerably easier to realize in that the sensors and actuators themselves are software. The cost is the errors involved in the modeling of the sensors and actuators and not being able to test the physical hardware. It is easy to identify the inherent trade-off between the detail and realism offered by the simulator and the scope needed to realize it.

A further trade-off might be to rid away with the A/D converters entirely and emulate that part of the embedded software, bypassing hardware entirely and communicating directly with the simulator over a suitable higher-level interface. On a microcontroller this interface might typically be a serial port to a desktop computer running the simulator. As long as the rest of the control system runs as is on the embedded platform, it is still considered hardware-in-the-loop simulation, albeit in a weak form, where all hardware except the computing platform is simulated in addition to a small part of the software. In contrast, if the control system code is running on the same desktop computer as the simulator, forgoing the serial connection and execution on a microcontroller entirely, we have a so-called *software-in-the-loop* simulation[13]. We will use this as a starting point when motivating the design of a simulator to aid the development of navigation, guidance and control algorithms for an autonomous flying robot.

1.2 Motivation

The work documented in this thesis is specifically motivated by work done with the student startup *Versor* developing an autopilot for an autonomous flying robot relying on sensor fusion of camera images and inertial measurements to navigate and range measurements from a lidar to map the surroundings. In particular it was discovered that testing code on real in-the-air hardware would create too much overhead and become non-effective in the long run. And the need for a simulator to test code as much as possible on the desktop during development, became apparent.

Ready-made robot simulators exist[10] and are in active development[16], and provide proof that the need for such a tool is real. However, creating an in-house solution from scratch offers many benefits: i) complete control of the physics engine and the generated sensor data, ii) customization by design without the need to develop peripheral plugins, and most importantly iii) effortless integration with the autopilot system, with adherence to the same proprietary communication system without the need for a hack to bridge gaps between communication mediums.

Figure 1.1 diagrams the overall autopilot system, in a simple form. In essence it consists of four main modules, navigation, guidance, control and input/output (IO), all running on Linux and communicating through socket based messaging. The IO module has a high-level interface connection, USB and Ethernet, to cameras, laser sensors and a microcontroller *IO-board* providing low-level electrical interfacing with actuators and inertial sensors. The microcontroller additionally performs high-frequency, low-latency closed-loop control of the actuators, so a small portion of the control codes actually resides outside Linux.

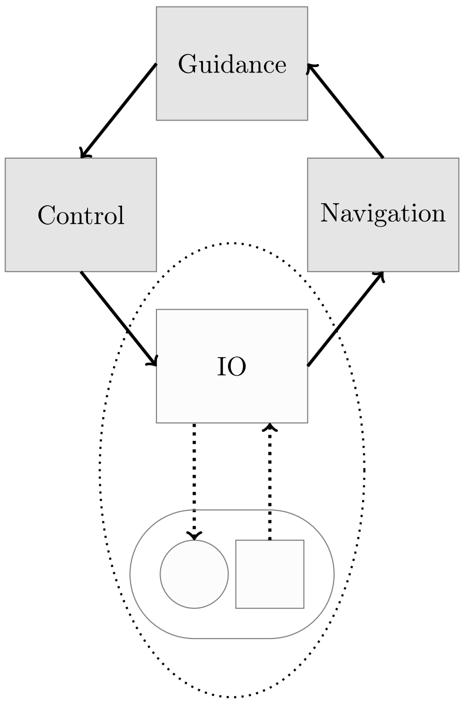


Figure 1.1: Simplified autopilot architecture. The dotted line indicates the part replaced by simulation.

In light of the previous discussion on hardware-in-the-loop simulation during embedded development, the question is what should be the scope of the simulator - which part of the overall system, if any, should it replace and emulate in addition to actuator response and sensor data. It seems interfacing with the IO-board at all is overly complex: Interfacing at the actual electronic level involves hardware and A/D converters. Additionally, the complexities and error modes of direct communication with the actual sensors will not and can not be modeled in the simulation, so the code responsible for the interface cannot be said to be sufficiently tested with such a simulation. Alternatively, given that the complexity of the simulation involved necessitates a desktop implementation, interfacing the simulator with the microcontroller board through a serial interface is not default operation for the embedded code, and changing it to fit the purpose of the simulation seems to void the entire purpose of the exercise which is to test the real code *as is*.

The question is then if the Linux IO module should be left unchanged. While it is relatively easy for the desktop simulator to communicate through network sockets and serial ports, this again seems overly complex as it requires spoofing of particular sensor protocols byte for byte, bit by bit, in order to test the fairly trivial code, compared to the rest of the system, which handles the sensor interfaces. One reason for this is that interfacing with the sensors is straight-forward to test in a desktop setting, without a full-scale demonstration in order to fly. Testing code correctness here does then not necessarily warrant the overhead of a simulator. Not to mention the fact that again the actual error modes of the sensors will remain untested anyway.

It seems the remaining reasonable alternative is to simulate the entire IO module and provide the same high-level sensor data to the navigation module in response to control input as a result of the vehicle dynamics. The modules are in their final form embedded on a system on a module running Linux. Aside from cross compilation to allow for differing architectures, in principle there is little difference between the code running on the embedded platform or on the desktop, and the network based inter-module communications would facilitate both equally. If we identify the embedded case as a hardware-in-the-loop simulation and the desktop case as software-in-the-loop, then it is clear that running embedded Linux blurs the definition between hardware and software-only in the loop, and makes the distinction in this case redundant. In either case, what is achieved is a perfectly transparent simulation in which the real part of the system not simulated does not feel the difference between operation in the real world and the simulated world. Given this proposed scope of the simulator we will describe in more detail the exact responsibilities required from the implementation.

1.3 Objectives

The main objectives of this work is to

1. Implement a headless simulator as a drop-in replacement of the input/output module to interact with the rest of the system in 1.1 *as is*. It should be performant enough to execute in real-time[1]

[1]Throughout we will use the term *real-time* to mean satisfactory average execution time in order to inspect results live, and not as it relates to low-latency response times to satisfy any strict real-time demands.

2. Generate synthetic sensor data, including
    - Inertial measurements consisting of acceleration and angular velocity. The measurement should emulate the noise characteristics of a specific IMU.
    - High frame rate camera images generated according to the intrinsic calibration parameters of a specific camera. The optical distortion effects of a wide-angle fisheye-like lense will have to be synthesized.
    - Range measurements from a scanning lidar. The generated data should replicate the field of view, range limitations and sampling frequency and density of a specific laser scanner.
3. Simulate general six-degrees of freedom rigid body dynamics as a result of a set of force and torque actuation inputs.
4. Model the force and torque actuation of a general quadrotor body in response to control input.
5. Develop visualization tools to interface with the headless simulator and the rest of the system, for live visual verification of relevant system states.

1.4 Contributions

The contributions of this thesis are considered to be

- The simulator itself. As the implementation source code itself will be kept proprietary, the contribution of this thesis is to uncover the basic modeling principles and implementation details.
- The derivation of relating proper acceleration to an alternative observer frame in order to correctly evolve terrestrial motion, and in particular the suggestion not to include gravity as a fictitious force when solving the Newton-Euler equation.
- The outlined method of generating an OpenGL projection matrix from intrinsic pinhole camera parameters.
- The presentation of the proposed duality between image distortion and undistortion methods, and the practical demonstration favouring the so-called indirect methods over direct methods.
- The method of frustum subdivision and a depth buffer sampling model to generate a point cloud emulating the measurements of a scanning lidar.

1.5 Outline

- **Chapter 2:** A fundamental study of rigid body dyhamics begins with describing the basic differential kinematic equations of motion. The results will be used repeatedly, including to derive the body-centric Newton-Euler equation of motion describing the relationship between forces and torques acting on a rigid body and the resulting accelerations.

- **Chapter 3:** A simple model of quadrotor actuation to describe the forces and torques acting on the body will be derived based on the configuration of the rotor actuators and their response to a control input.
- **Chapter 4:** Based on a measurement model for accelerometers and gyroscopes, a set of inertial measurements are generated in accordance with the simulated body dynamics.
- **Chapter 5:** Using graphics programming and the concept of an OpenGL camera, camera images of a 3D scene are constructed, replicating the perspective of a specific camera given its intrinsic calibration parameters. An effort is made to replicate the optical distortion effects of a wide-angle lense, and in particular the duality between image distortion and undistortion methods is presented.
- **Chapter 6:** The same graphical programming techniques are used to generate a set of ideal laser range measurements of a 3D scene to simulate the expected data from a typical scanning lidar.
- **Chapter 7:** Implementation specifics are described with a focus on detailing numerical values for modeling parameters, in relation to the specific sensor components.
- **Chapter 8:** The qualitative results from the simulation are presented through visualization, and the validity of the simulation and its role in ongoing software development is discussed.
- **Chapter 9:** Conclusion and further work.

2 Rigid Body Dynamics

We will derive the basic equations of motion for rigid body systems, as a means to simulate the movement of a body in response to external forces. A rigid body does not deform, and so the analysis reduces to describing the change in position and orientation of the body frame over time. The material is in large part based on *Modeling and Simulation for Automatic Control*[8] (Egeland and Gravdahl 2003) but an independent effort has been made to derive key equations which are otherwise not derived or derived using coordinate free vectors.

2.1 Kinematics

Kinematics studies the geometry of motion. It describes the possible motions of rigid body configurations, without concern for forces or torques necessary to cause the motion. The important equation derived in this section is (2.5), which relate the acceleration of a point in one coordinate frame to another.

2.1.1 Coordinate frames

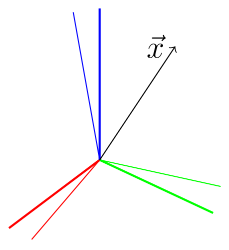


Figure 2.1: Vector $\vec{x}$, and unit axes ($\{\vec{i}, \vec{j}, \vec{k}\} \mapsto \{\text{red}, \text{green}, \text{blue}\}$) of coordinate frames a (*thick*) and b (*thin*).

A vector is a geometrical object with a magnitude and direction. It can be graphically represented as an arrow pointing from A to B, as $\vec{x}$ in figure 2.1. This vector does not care what coordinate system is used to describe it, it is *coordinate free*, and cares only about the laws of vector algebra. However, to describe the vector numerically, we need to represent its length and direction in relation to some system of coordinates, a *coordinate frame*. In such a state we have a *coordinate vector* described in terms of the coordinate frame basis. Defining two Cartesian coordinate frames a and b, with orthogonal unit axes vectors $\vec{a}_0$, $\vec{a}_1$, $\vec{a}_2$ and $\vec{b}_0$, $\vec{b}_1$, $\vec{b}_2$, respectively, a vector $\vec{x}$ is resolved in turn as

$$\mathbf{x}^a = \begin{pmatrix} x_0^a \\ x_1^a \\ x_2^a \end{pmatrix} \quad \mathbf{x}^b = \begin{pmatrix} x_1^b \\ x_1^b \\ x_2^b \end{pmatrix}$$

where $x_i^a = \vec{x} \cdot \vec{a}_i$ and $x_i^b = \vec{x} \cdot \vec{b}_i$. The frames are meant to be identical except for their orientation relative to $\vec{x}$. Their relative orientation is described by the rotation matrix[8, (6.83)] $\mathbf{R}_b^a$ where $(\mathbf{R}_b^a)_{ij} = \vec{a}_i \cdot \vec{b}_j$ such that

$$\mathbf{x}^a = \mathbf{R}_b^a \mathbf{x}^b \tag{2.1}$$

Throughout we will use the rotation matrix $\mathbf{R}_b^a$ to mean a passive coordinate frame transform from b to a, never an active rotation of a vector within a frame. The frame b is considered a *local* frame with respect to a and conversely a as *global* with respect to b. This is the passive, local-to-global convention as described in [25]. To describe the inverse transformation, from a to b we will use $\mathbf{R}_b^{a\top}$.

2.1.2 Frames of reference

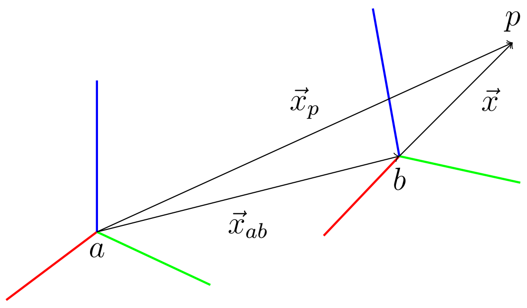


Figure 2.2: Configuration of two arbitrarily moving reference frames, a and b, and point p.

So far coordinate frames have only differed in their relative orientations. We can extend this by adding relative positioning between frames to develop the more general concept of a coordinate *frame of reference.*

Consider the relative position $\vec{x}_{ab}$ from the center of frame a to the center of frame b. It can be resolved in both frames, respectively $\mathbf{x}_{ab}^a$ and $\mathbf{x}_{ab}^b$, and for any vector $\mathbf{x}^b$ in b pointing to an arbitrary point p, the vector to the same point in a is

$$\mathbf{x}_p^a = \mathbf{x}_{ab}^a + \mathbf{R}_b^a \mathbf{x}^b \tag{2.2}$$

The relationship can be described with a matrix operation involving homogeneous coordinates, a *homogeneous transform*

$$\begin{pmatrix} \mathbf{x}_p^a \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_b^a & \mathbf{x}_{ab}^a \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x}^b \\ 1 \end{pmatrix} \tag{2.3}$$

2.1.3 Differential equations

To study the dynamic relationship between coordinate vectors, time differentiation of (2.1) gives

$$\begin{aligned} \dot{\mathbf{x}}^a &= \frac{d}{dt}\left(\mathbf{R}_b^a \mathbf{x}^b\right) \\ &= \mathbf{R}_b^a \dot{\mathbf{x}}^b + \dot{\mathbf{R}}_b^a \mathbf{x}^b \end{aligned}$$

Based on the orthogonal properties of the rotation matrix it is easily shown that [8, (6.225)] $\dot{\mathbf{R}}_b^a = \mathbf{R}_b^a (\boldsymbol{\omega}_{ab}^b)^\times$ where $(\boldsymbol{\omega}_{ab}^b)^\times$ is the skew symmetric matrix of the so-called angular velocity vector $\boldsymbol{\omega}_{ab}^b$ of frame b relative to frame a resolved in, or *seen from*, b. Simple substitution then leads to the basic kinematic equation

$$\dot{\mathbf{x}}^a = \mathbf{R}_b^a \left(\dot{\mathbf{x}}^b + (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{x}^b\right) \tag{2.4}$$

as given by [8, (6.275)], relating the time derivative of $\vec{x}$ as seen from b to the time derivative of $\vec{x}$ as seen from a. As shown in figure 2.2, we can combine (2.4) with (2.2) to relate the velocity of some arbitrarily moving point p between reference frames a and b which are also moving arbitrarily:

$$\dot{\mathbf{x}}_p^a = \dot{\mathbf{x}}_{ab}^a + \mathbf{R}_b^a \left(\dot{\mathbf{x}}^b + (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{x}^b\right)$$

Again, using (2.4), further time differentiation gives us the acceleration of point p seen from a, given the acceleration of p as seen from b:

$$\begin{aligned} \ddot{\mathbf{x}}_p^a &= \ddot{\mathbf{x}}_{ab}^a + \mathbf{R}_b^a \left[\frac{d}{dt}\left(\dot{\mathbf{x}}^b + (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{x}^b\right) + (\boldsymbol{\omega}_{ab}^b)^\times \left(\dot{\mathbf{x}}^b + (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{x}^b\right)\right] \\ &= \ddot{\mathbf{x}}_{ab}^a + \mathbf{R}_b^a \left[\ddot{\mathbf{x}}^b + (\dot{\boldsymbol{\omega}}_{ab}^b)^\times \mathbf{x}^b + 2(\boldsymbol{\omega}_{ab}^b)^\times \dot{\mathbf{x}}^b + (\boldsymbol{\omega}_{ab}^b)^\times (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{x}^b\right] \end{aligned}$$

as stated in [8, (6.404)]. This is a very important result, and will be used repeatedly in this work to relate the acceleration of a point between reference frames. For convenience, labeling $\mathbf{r}$ as position, $\mathbf{v}$ as linear velocity, $\mathbf{a}$ as linear acceleration, and $\boldsymbol{\alpha}$ as angular acceleration we get

$$\mathbf{a}_p^a = \mathbf{a}_{ab}^a + \mathbf{R}_b^a \left[\mathbf{a}^b + (\boldsymbol{\alpha}_{ab}^b)^\times \mathbf{r}^b + 2(\boldsymbol{\omega}_{ab}^b)^\times \mathbf{v}^b + (\boldsymbol{\omega}_{ab}^b)^\times (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{r}^b\right]$$

To reiterate, given the acceleration of a point p in b, $\mathbf{a}^b$, and an arbitrary acceleration and rotation between frames a and b, the equation returns the acceleration of the point in relation to, *and* resolved in, frame a: $\mathbf{a}_p^a$. However, it turns out to be convenient, and equally valid, to resolve the resulting acceleration in the b frame:

$$\mathbf{a}_p^b = \mathbf{a}_{ab}^b + \mathbf{a}^b + (\boldsymbol{\alpha}_{ab}^b)^\times \mathbf{r}^b + 2(\boldsymbol{\omega}_{ab}^b)^\times \mathbf{v}^b + (\boldsymbol{\omega}_{ab}^b)^\times (\boldsymbol{\omega}_{ab}^b)^\times \mathbf{r}^b \tag{2.5}$$

where $\mathbf{a}_p^b = (\mathbf{R}_b^a)^\top \mathbf{a}_p^a$ and $\mathbf{a}_{ab}^b = (\mathbf{R}_b^a)^\top \mathbf{a}_{ab}^a$[1]. (2.5) will be used to evolve the solution to the kinetic Newton-Euler equation in an arbitrarily moving reference frame of choice.

2.2 Kinetics

In contrast to kinematics, we will use the term *kinetics* to mean the study of motion in relation to its causes. The important result here is the derivation of the Newton-Euler equation of motion (2.6), relating the forces and torques acting on a rigid body to the resulting linear and angular accelerations in relation to an inertial frame of reference.

2.2.1 Inertial frames of reference

While reference frames in physics can be used to describe the position and orientation of an observer in space, a so called *inertial reference frame* is a more abstract concept. Inertial reference frames are a subset of all possible reference frames, in which none exhibit any linear acceleration or angular velocity in respect to each other or, by definition, in absolute terms. This means an accelerometer moving with an inertial frame would detect no acceleration, and a gyroscope would detect no angular velocity.

Physical laws take the same, simplest form in all inertial frames, as the physics of a system in an inertial frame have no causes external to the system[34]. In terms of Newton's second law, for example, this means there is no *proper*, i.e inertial, acceleration without an acting force, and vice versa. General relativity implies there are no true inertial frames around gravitating bodies. An accelerometer fixed to the surface of the Earth, in a non-inertial frame, would experience an upwards proper acceleration of about 1 g despite not undergoing any coordinate acceleration or movement in the coordinate frame. The only force acting on the body is the normal force from the ground stopping the body from free-falling along its inertial path to the center of the Earth. Gravity, then, does not cause proper acceleration and, in terms of general relativity, is not a force, just a consequence of the curvature of space-time. Due to, but not limited to, the curvature of space-time at the surface of the Earth, in contrast to Newton's second law there is a mismatch between the forces acting on the body and the experienced acceleration.

[1]Be aware that with this notation that $\mathbf{a}_p^b = (\mathbf{R}_b^a)^\top \ddot{\mathbf{x}}_p^b \neq \ddot{\mathbf{x}}_p^b$, and correspondingly for $\mathbf{a}_{ab}^b$.

As is stands, a coordinate frame fixed to the surface of the Earth is not a good representation of an inertial frame, and there are better approximations to be made. We will derive the Newton-Euler equation of motion in its true form relating to inertial space. However, for now it is sufficient to be aware of inertial space as a concept without need to define an approximate inertial frame of reference for absolute positioning.

2.2.2 Newton-Euler equation of motion

To derive the Newton-Euler equation of motion we start with Euler's extension of Newton's second law of physics[8, (7.32),(7.33)]:

$$\begin{aligned} \mathbf{f}^i_{bc} &= m\mathbf{a}^i_c \\ \boldsymbol{\tau}^i_{bc} &= \frac{d}{dt}\left(\mathbf{M}^i_{b/c}\boldsymbol{\omega}^i_{ib}\right) \end{aligned}$$

The first equation relates a force $\mathbf{f}^i_{bc}$ acting on the body b with a line of action through the mass center c and the resulting *proper* linear acceleration $\mathbf{a}^i_c$ of the mass center in inertial space. Notice that both terms are resolved in the inertial frame i, but the body-centric version $\mathbf{f}^b_{bc} = m\mathbf{a}^b_c$ is equivalent, and equally valid, as long as $\mathbf{a}^b_c$ is the same proper acceleration of the mass center, only resolved in the body frame, i.e $\mathbf{a}^b_c = (\mathbf{R}^i_b)^\top \mathbf{a}^i_c$ and $\mathbf{f}^b_{bc} = (\mathbf{R}^i_b)^\top \mathbf{f}^i_{bc}$

The second equation relates a torque $\boldsymbol{\tau}^i_{bc}$ equalling the total moment of the body around c to the time rate of change in linear momentum $\mathbf{M}^i_{b/c}\boldsymbol{\omega}^i_{ib}$. The Inertia tensor $\mathbf{M}^i_{b/c}$ of the body around c is not constant under rotation when resolved in an inertial frame, so it is not completely straight forward to derive the body-centric description. Introducing the similarity transform[8, (7.80)] $\mathbf{M}^i = \mathbf{R}^i_b\mathbf{M}^b_{bc}(\mathbf{R}^i_b)^\top$, where $\dot{\mathbf{M}}^b_{bc} = 0$ by definition, and using (2.4), we get

$$\begin{aligned} \frac{d}{dt}\left(\mathbf{M}^i_{b/c}\boldsymbol{\omega}^i_{ib}\right) &= \frac{d}{dt}\left(\mathbf{R}^i_b\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}\right) \\ &= \mathbf{R}^i_b\left(\frac{d}{dt}\left(\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}\right) + (\boldsymbol{\omega}^b_{ib})^\times\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}\right) \\ &= \mathbf{R}^i_b\left(\dot{\mathbf{M}}^b_{b/c}\boldsymbol{\omega}^b_{ib} + \mathbf{M}^b_{b/c}\dot{\boldsymbol{\omega}}^b_{ib} + (\boldsymbol{\omega}^b_{ib})^\times\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}\right) \\ &= \mathbf{R}^i_b\left(\mathbf{M}^b_{b/c}\dot{\boldsymbol{\omega}}^b_{ib} + (\boldsymbol{\omega}^b_{ib})^\times\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}\right) \end{aligned}$$

The torque in body frame $\boldsymbol{\tau}^b_{bc} = (\mathbf{R}^i_b)^\top\boldsymbol{\tau}^i_{bc}$ is then

$$\boldsymbol{\tau}^b_{bc} = \mathbf{M}^b_{b/c}\boldsymbol{\alpha}^b_{ib} + (\boldsymbol{\omega}^b_{ib})^\times\mathbf{M}^b_{b/c}\boldsymbol{\omega}^b_{ib}$$

The results is the body-centric Newton-Euler equations of motion[8, (7.90)], expressed compactly in a six-dimensional matrix equation as

$$\begin{pmatrix} m\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{M}^b_{b/c} \end{pmatrix} \begin{pmatrix} \mathbf{a}^b_c \\ \boldsymbol{\alpha}^b_{ib} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ (\boldsymbol{\omega}^b_{ib})^\times \mathbf{M}^b_{b/c} \boldsymbol{\omega}^b_{ib} \end{pmatrix} = \begin{pmatrix} \mathbf{f}^b_{bc} \\ \boldsymbol{\tau}^b_{bc} \end{pmatrix} \tag{2.6}$$

Notice that the result does not explicitly refer to any absolute location and orientation of any inertial reference frame, and only linear accelerations and angular velocity. The result of (2.6) expects a body frame centered in the center of mass. The more general result takes into consideration a given offset $\mathbf{r}^b$ from body center o to mass center c. Then, using (2.5), where $\mathbf{v}^b = \dot{\mathbf{r}}^b = 0$, $\mathbf{a}^b = \ddot{\mathbf{r}}^b = 0$

$$\mathbf{a}^b_c = \mathbf{a}^b_o + (\boldsymbol{\alpha}^b_{ib})^\times \mathbf{r}^b + (\boldsymbol{\omega}^b_{ib})^\times (\boldsymbol{\omega}^b_{ib})^\times \mathbf{r}^b \tag{2.7}$$

where $\mathbf{a}^b_o$ is the proper acceleration of the body frame center. Solving for $\mathbf{a}^b_o$ the force equation becomes

$$\mathbf{f}^b_{bo} = m \left(\mathbf{a}^b_o + (\boldsymbol{\alpha}^b_{ib})^\times \mathbf{r}^b + (\boldsymbol{\omega}^b_{ib})^\times (\boldsymbol{\omega}^b_{ib})^\times \mathbf{r}^b \right)$$

where[8, (7.17)] $\mathbf{f}^b_{bo} = \mathbf{f}^b_{bc}$ is the resultant force vector with line of action through o. For the torque $\boldsymbol{\tau}^b_{bo}$ to equal the total moment of the body around o we have[8, (7.18)] $\boldsymbol{\tau}^b_{bo} = \boldsymbol{\tau}^b_{bc} + (\mathbf{r}^b)^\times \mathbf{f}^b_{bc}$ where $\boldsymbol{\tau}^b_{bc}$ and $\mathbf{f}^b_{bc}$ are given by (2.6) in terms of accelerations. We will use (2.7) and the parallel axes theorem[8, (7.85)] $\mathbf{M}^b_{b/o} = \mathbf{M}^b_{b/c} - m(\mathbf{r}^b)^\times (\mathbf{r}^b)^\times$ to derive the very pleasing result of [8, (7.95)]:

$$\begin{aligned}
\boldsymbol{\tau}^b_{bo} &= \boldsymbol{\tau}^b_{bc} + (\mathbf{r}^b)^\times \mathbf{f}^b_{bc} \\
&= (\boldsymbol{\omega}^b_{ib})^\times \mathbf{M}^b_{b/c} \boldsymbol{\omega}^b_{ib} + \mathbf{M}^b_{b/c} \boldsymbol{\alpha}^b_{ib} + m(\mathbf{r}^b)^\times \mathbf{a}^b_c \\
&= \ (\boldsymbol{\omega}^b_{ib})^\times \mathbf{M}^b_{b/o} \boldsymbol{\omega}^b_{ib} + \mathbf{M}^b_{b/o} \boldsymbol{\alpha}^b_{ib} + m(\mathbf{r}^b)^\times \mathbf{a}^b_o \\
&\quad + m(\boldsymbol{\omega}^b_{ib})^\times (\mathbf{r}^b)^\times (\mathbf{r}^b)^\times \boldsymbol{\omega}^b_{ib} + m(\mathbf{r}^b)^\times (\mathbf{r}^b)^\times \boldsymbol{\alpha}^b_{ib} \\
&\quad + m(\mathbf{r}^b)^\times (\boldsymbol{\omega}^b_{ib})^\times (\boldsymbol{\omega}^b_{ib})^\times \mathbf{r}^b + m(\mathbf{r}^b)^\times (\boldsymbol{\alpha}^b_{ib})^\times \mathbf{r}^b \\
&= (\boldsymbol{\omega}^b_{ib})^\times \mathbf{M}^b_{b/o} \boldsymbol{\omega}^b_{ib} + \mathbf{M}^b_{b/o} \boldsymbol{\alpha}^b_{ib} + m(\mathbf{r}^b)^\times \mathbf{a}^b_o
\end{aligned}$$

The last identity first follows from the cross product anti-commutative property $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$ and distributive property $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$ giving

$$(\mathbf{r}^b)^\times (\mathbf{r}^b)^\times \boldsymbol{\alpha}^b_{ib} + (\mathbf{r}^b)^\times (\boldsymbol{\alpha}^b_{ib})^\times \mathbf{r}^b = (\mathbf{r}^b)^\times \left((\mathbf{r}^b)^\times \boldsymbol{\alpha}^b_{ib} + (\boldsymbol{\alpha}^b_{ib})^\times \mathbf{r}^b \right) = 0$$

Secondly, the skew symmetric identity[8, (6.33)] $(\mathbf{a}^\times \mathbf{b})^\times = \mathbf{a}^\times \mathbf{b}^\times - \mathbf{b}^\times \mathbf{a}^\times$ and the cross product identity $\mathbf{a} \times \mathbf{a} = 0$ gives

$$
\begin{aligned}
&(\boldsymbol{\omega}_{ib}^b)^\times(\mathbf{r}^b)^\times(\mathbf{r}^b)^\times\boldsymbol{\omega}_{ib}^b + (\mathbf{r}^b)^\times(\boldsymbol{\omega}_{ib}^b)^\times(\boldsymbol{\omega}_{ib}^b)^\times\mathbf{r}^b \\
&= \left((\boldsymbol{\omega}_{ib}^b)^\times(\mathbf{r}^b)^\times - (\mathbf{r}^b)^\times(\boldsymbol{\omega}_{ib}^b)^\times\right)(\mathbf{r}^b)^\times\boldsymbol{\omega}_{ib}^b \\
&= -\left((\boldsymbol{\omega}_{ib}^b)^\times(\mathbf{r}^b)\right)^\times(\boldsymbol{\omega}_{ib}^b)^\times\mathbf{r}^b \\
&= 0
\end{aligned}
$$

Using the defining skew symmetric propetry $(\mathbf{r}^b)^\times = -(\mathbf{r}^b)^{\times\top}$ the end result is the general body-centric Newton-Euler equation of motion[8, (7.96)]

$$
\begin{pmatrix} m\mathbf{I} & m(\mathbf{r}^b)^{\times\top} \\ m(\mathbf{r}^b)^\times & \mathbf{M}_{b/o}^b \end{pmatrix}\begin{pmatrix} \mathbf{a}_o^b \\ \boldsymbol{\alpha}_{ib}^b \end{pmatrix} + \begin{pmatrix} m(\boldsymbol{\omega}_{ib}^b)^\times(\boldsymbol{\omega}_{ib}^b)^\times\mathbf{r}^b \\ (\boldsymbol{\omega}_{ib}^b)^\times\mathbf{M}_{b/o}^b\boldsymbol{\omega}_{ib}^b \end{pmatrix} = \begin{pmatrix} \mathbf{f}_{bo}^b \\ \boldsymbol{\tau}_{bo}^b \end{pmatrix} \tag{2.8}
$$

Notice that with $\mathbf{r}^b = 0$, the general case reverts to the simpler (2.6). Solving (2.8) for the accelerations vector, we get

$$
\begin{pmatrix} \mathbf{a}_o^b \\ \boldsymbol{\alpha}_{ib}^b \end{pmatrix} = \begin{pmatrix} m\mathbf{I} & m(\mathbf{r}^b)^{\times\top} \\ m(\mathbf{r}^b)^\times & \mathbf{M}_{b/o}^b \end{pmatrix}^{-1}\begin{pmatrix} \mathbf{f}_{bc}^b \\ \boldsymbol{\tau}_{bc}^b \end{pmatrix} - \begin{pmatrix} m(\boldsymbol{\omega}_{ib}^b)^\times(\boldsymbol{\omega}_{ib}^b)^\times\mathbf{r}^b \\ (\boldsymbol{\omega}_{ib}^b)^\times\mathbf{M}_{b/o}^b\boldsymbol{\omega}_{ib}^b \end{pmatrix} \tag{2.9}
$$

where the so-called *inertia matrix*[2] is decidedly invertible[11]. Notice that when $\mathbf{r}^b = 0$, the decoupling of the linear and angular acceleration of (2.8) is easier to solve in that only the inertia tensor needs to be inverted.

2.3 Navigational kinematics

Observe that the result of solving the Newton-Euler equation is the inertial linear and angular accelerations. Integrating these results directly leads to a change in position and orientation in an inertial reference frame. This poses a practical problem, and ideally we would like to integrate the solution in an alternative *observational* frame of reference.

2.3.1 Inertial reference frame approximations

As mentioned when introducing inertial reference frames, no inertial frames exist near gravitational bodies. Consider a so-called Earth-centered inertial frame (ECI), like the Geocentric Celestial Reference Frame, non-rotating and fixed at the center of the Earth. The Earth is free-falling around the Sun and an ECI is considered inertial with good approximation sufficient for many applications, like satellites orbiting the Earth[28]. At least on a shorter time scale where the interplay from gravitational attraction of other heavenly bodies such as the Sun and the Moon causing secondary gravitational effects like precession and nutation of the Earths rotational axis can be neglected. It is important

[2]Not to be confused with the inertia tensor $\mathbf{M}_{b/o}^b$.

to note that an ECI is only approximately inertial at a small region of space around its center, where space-time is approximately flat. For applications in space involving orbital motion it makes sense to employ an ECI, as orbital objects free falls around the Earth and does not experience any acceleration relative to the reference frame. Additionally the equations of motion for orbital motion are simpler in a non-rotating frame.

For terrestrial applications, the opposite is true. An object fixed to the surface of the Earth would experience a proper acceleration of about $1\,g$ upwards. Employing an approximate inertial frame of reference like an ECI does in itself nothing to remedy this as its inertial assumption is only valid near the center of the Earth or in free fall. This means for terrestrial applications the effects of gravity will have to be correctly compensated for, regardless if an ECI is used or not. Setting aside the effect of gravity, we will assume any given ECI is sufficiently inertial for our purposes. However, it seems unnecessarily complicated to relate terrestrial motion to a non-rotating frame fixed to distant stars. A geocentric view employing an Earth-fixed frame rotating with the Earth is more convenient.

The Earth rotates around its own axis once per day. This results in a very small angular rate and its reasonable to ask whether or not it can be ignored for our purposes, and let the Earth-fixed frame be considered sufficiently inertial. What should be considered sufficiently inertial depends on the technology involved in the application and the sensitivity of the instruments[28]. For our purposes we rely on simulating gyroscopic data for sensitive inertial measurement units that can detect this kind of angular rate. We decide it is a worthwhile effort not to rely on the Earth-fixed frame as sufficiently inertial, and to include the effects of the rotation of the Earth around its own axis. Using an alternative frame as an observational frame of reference means we do not have to define and relate to a particular ECI, but we have to take care when integrating the solution to the Newton-Euler equation, as we shall see.

2.3.2 Earth-centered, Earth-fixed reference frame

When deciding on an Earth-fixed observational frame of reference it is clear that an Earth-centered one offers many advantages. It shares a center with an ECI and experience only relative angular velocity, no linear velocity. The result is that the kinematic equations relating inertial and Earth-fixed accelerations are easy to express in the Earth-centered, Earth-fixed (ECEF) case. Additionally, a geocentric Earth-centered frame offers a convenient point of reference for global terrestrial movement. For now, assume an ECEF frame, called the Earth frame e is defined. Let i denote an arbitrary ECI frame considered sufficiently inertial. Given a body frame located at $\mathbf{r}_b^e$ in the Earth frame and a solution $\mathbf{a}_o^b$ of (2.9), it is clear from (2.5), the acceleration of the body in the Earth frame[3] is

$$\mathbf{a}_b^e = \underbrace{\mathbf{R}_b^e \mathbf{a}_o^b}_{\text{Proper acc.}} - \underbrace{\mathbf{a}_{ie}^e}_{\text{g-force}} - \underbrace{(\boldsymbol{\alpha}_{ie}^e)^\times \mathbf{r}_b^e}_{\text{Euler acc.}} - \underbrace{2(\boldsymbol{\omega}_{ie}^e)^\times \mathbf{v}_b^e}_{\text{Coriolis acc.}} - \underbrace{(\boldsymbol{\omega}_{ie}^e)^\times (\boldsymbol{\omega}_{ie}^e)^\times \mathbf{r}_b^e}_{\text{Centripetal acc.}} \tag{2.10}$$

[3]This is only applied to linear acceleration, as angular acceleration is integrated to angular velocity directly in its inertial form.

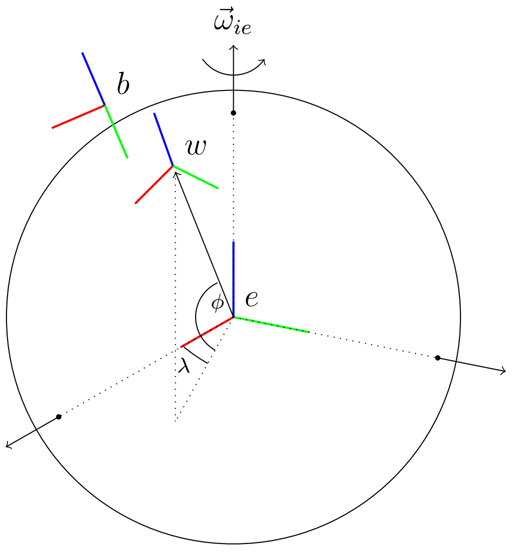


Figure 2.3: Global ECEF Earth frame e, local navigational frame w, and body frame b.

Assuming the Earth rotates with a fixed angular rate, then $\boldsymbol{\alpha}_{ie}^{e} = 0$. While e remain fixed at the center of Earth and not strictly accelerating away from the center it might be tempting to allow $\mathbf{a}_{ie}^{e} = 0$ as well. But due to the effects of the curvature of space-time around the Earth, gravity has to be compensated for. Therefore, the correct approach it to set what we call the g-force $\mathbf{a}_{ie}^{e}$ equal to the *negative* gravity at $\mathbf{r}_{b}^{e}$, as if the e frame is accelerating away from the free-fall trajectory of an inertial frame towards the center of the Earth. The remaining detail is to only let $\mathbf{a}_{ie}^{e}$ influence the movement of the rigid body, presumably above Earth's topographical surface and far away from the center, while the Earth frame itself remains stationary in the center. Computationally, this is exactly what we want, but for semantic consistency we have to concede that $\mathbf{a}_{ie}^{e} \neq 0$ does not result in actual relative movement between the Earth frame and the Earth center.

The more conventional approach is to include gravity multiplied with the body mass as a pseudo force on the right hand side of (2.8). This would be equivalent but with the side effect that you do not get the proper, inertial acceleration when solving the equation, but one including acceleration due to gravity. The proper acceleration is required when for example describing accelerometer measurements. Without the proper acceleration at hand there is a need to subtract gravity from the non-proper description of acceleration to achieve the wanted result. This is seen in [8, p. 258], [25, (215)], [24, (4.1)]. To define gravity in respect to a position $\mathbf{r}_{b}^{e}$ we have to more accurately describe the Earth frame. The Geodetic System 1984 (WSG84) defines a global geocentric reference frame, an ellipsoid and geodetic reference surface for the Earth, and gravity models based on these descriptions. WGS84 represents the best global geodetic reference system available for the Earth at this time, for practical geopositioning and navigation[3].

The WGS84 reference coordinate frame is defined in the mass center of the Earth, with a z-axis pointing in the direction of the IERS (International Earth Rotation and Reference Systems Service) Reference Pole, an x-axis intersecting the IERS Reference Meridian and the equatorial plane normal to the z-axis passing through the origin, and lastly a y-axis completing a right-handed ECEF coordinate frame. The coordinate frame origin also serves as the geometric center of the WGS84 Reference Ellipsoid of revolution (a spheroid), where the z-axis serves as the rotational axis of revolution. If the Earth to a first-order approximation is a rotating sphere, to a second-order approximation it can be regarded as an equipotential ellipsoid of revolution. The ellipsoid serves as a uniform reference surface for geophysical purposes, such as map projections and satellite navigation. In addition it serves as the reference surface for the normal gravity model of the Earth.

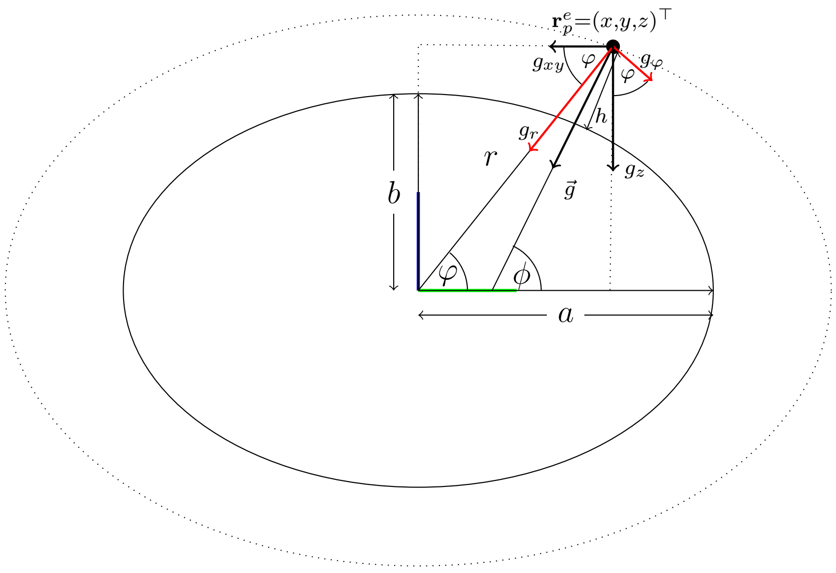


Figure 2.4: Reference ellipsoid and normal gravity model. Ellipsoidal form exaggerated for effect. Theoretical gravity $\vec{g}$ normal to the ellipsoid. Decomposed in spherical components g_r, g_φ or rectangular components g_{xy}, g_z.

The WGS84 ellipsoid is defined as an equipotential surface with a specific theoretical gravity potential U which is independent of the density distribution within the ellipsoid and uniquely determined with the four defining parameters of the ellipsoid given in table 7.1. The theoretical normal gravity model defines the gravitational magnitude γ_0 of the gradient of the potential function U at the surface of the ellipsoid, for a given geodetic latitude ϕ[3, (4-1)]:

$$\gamma_0(\phi) = \gamma_e \frac{1 + k\sin^2\phi}{\sqrt{1 - e^2\sin^2\phi}}$$

where $k = \frac{b\gamma_p}{a\gamma_e} - 1$, $e^2 = 1 - \frac{b^2}{a^2}$ for the normal gravity γ_e at equator, normal gravity γ_p at the poles, semi-major axis a and semi-minor axis b as defined in [3]. The gravitational field of an ellipsoid is of great practical value as it is mathematically simple, and yet the deviations from the theoretical normal field to the actual gravity field are small. The normal gravity model can be extended to a closed form solution for any point below or above the ellipsoid. Equations [3, (4-5)–(4-20)] define g_r, g_φ as the spherical components of the normal gravity corresponding to a point P given by $\mathbf{r}_p^e = (x, y, z)^\top$, in geodetic ellipsoidal coordinates as $(\lambda, \phi, h)^\top$ and in geocentric spherical coordinates as $(\lambda, \varphi, r)^\top$, as shown in figure 2.4. Projecting the spherical components onto our wanted rectangular ECEF coordinates, we get

$$\begin{aligned} g_x &= g_{xy} \cos\lambda \\ g_y &= g_{xy} \sin\lambda \\ g_z &= g_r \sin\varphi + g_\varphi \cos\varphi \end{aligned}$$

where $g_{xy} = g_r \cos\varphi - g_\varphi \sin\varphi$ is the projection onto the equatorial plane, and $\lambda = \arctan 2(y, x)$, $\varphi = \arctan 2(z, \sqrt{x^2 + y^2})$ are the longitude and geocentric latitude, respectively. The resulting gravity vector in point P is then $\mathbf{g}_p^e = (g_x, g_y, g_z)^\top$.

2.3.3 Local-level navigational reference frame

While the Earth frame is a practical means to relate global positioning, in many cases relating to a reference frame for more local navigation is convenient. We define here the local reference frame, the world frame w, fixed to the Earth on or near its surface as shown in figure 2.3. It will remain static in the Earth frame and defined by its position $\mathbf{r}_w^e$ and orientation $\mathbf{R}_w^e$. To define the pose it is very convenient to use the reference ellipsoid and normal gravity model.

The position $\mathbf{r}_w^e$ is aptly described by a geodetic longitude and latitude and a height above the ellipsoid. For a given ellipsoidal point $(\lambda, \phi, h)^\top$ the corresponding rectangular coordinates are given by [3, (4-14),(4-15)]:

$$\begin{aligned} x &= (N + h) \cos\phi \cos\lambda \\ y &= (N + h) \cos\phi \sin\lambda \\ z &= (N(1 - e^2) + h) \sin\phi \end{aligned}$$

where $N = \frac{a}{\sqrt{1 - e^2 \sin^2\phi}}$ is the radius of curvature in the prime vertical, and $e^2 = 1 - \frac{b^2}{a^2}$ as before. Then $\mathbf{r}_w^e = (x, y, z)^\top$. The orientation $\mathbf{R}_w^e$ can be set with the z-axis normal to the ellipsoidal surface, with *yaw* as a free parameter, by applying the rotation between the local gravitation $\mathbf{g}^w = (0, 0, -1)^\top$ normalized for convenience, and the global normal gravitation $\mathbf{g}_w^e$ at the origin. This is outlined also in section 7.1. It is convenient to initialize a rigid body in terms of its local world coordinates. Given a wanted initial state $\mathbf{r}_{wb}^w$, $\mathbf{R}_b^w$ and velocities $\mathbf{v}_b^e$, $\boldsymbol{\omega}_{wb}^b$ we have

$$\begin{aligned} \mathbf{r}_b^e &= \mathbf{r}_w^e + \mathbf{R}_w^e \mathbf{r}_{wb}^w & \qquad \mathbf{R}_b^e &= \mathbf{R}_w^e \mathbf{R}_b^w \\ \mathbf{v}_b^e &= \mathbf{R}_w^e \mathbf{v}_b^w & \qquad \boldsymbol{\omega}_{ib}^b &= (\mathbf{R}_b^e)^\top \boldsymbol{\omega}_{ie}^e + \boldsymbol{\omega}_{wb}^b \end{aligned}$$

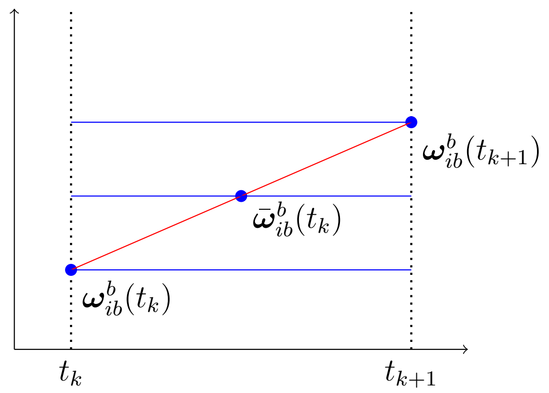
where obviously $\boldsymbol{\omega}_{ew}^b = \mathbf{0}$. Conversely, to transform back to the local frame, to observe a global Earth coordinate, we have

$$\mathbf{r}^w_{wb} = (\mathbf{R}^e_w)^\top(\mathbf{r}^e_b - \mathbf{r}^e_w) \qquad \mathbf{R}^w_b = (\mathbf{R}^e_w)^\top\mathbf{R}^e_b$$
$$\mathbf{v}^w_b = (\mathbf{R}^e_w)^\top\mathbf{v}^e_b \qquad \boldsymbol{\omega}^b_{wb} = \boldsymbol{\omega}^b_{ib} - (\mathbf{R}^e_b)^\top\boldsymbol{\omega}^e_{ie}$$

2.4 Numerical solution

From the accelerations vector $(\mathbf{a}^e_b, \boldsymbol{\alpha}^b_{ib})^\top$ derived from (2.10) after solving (2.9) we want to evolve the velocity, position and orientation of the body over time and keep track of the state $\mathbf{x}_b = \left(\mathbf{r}^e_b, \mathbf{v}^e_b, \mathbf{q}^e_b, \boldsymbol{\omega}^b_{ib}\right)^\top$. It is important to keep track of the angular velocity relative inertial space as it is required to solve (2.8). Although rotation matrices are used throughout to mathematically represent orientation transformation for reasons of simplicity, quaternion representations are more numerically efficient and favored during implementation[4]. We will deal with a mathematical description of quaternions here only, as it pertains to quaternion integration from angular velocity. The same passive, local-to-global convention described in [25] is used, meaning that $\mathbf{x}^a = \mathbf{R}^a_b\mathbf{x}^b$ and $\begin{pmatrix} 0 \\ \mathbf{x}^a \end{pmatrix} = \mathbf{q}^a_b \otimes \begin{pmatrix} 0 \\ \mathbf{x}^b \end{pmatrix} \otimes \tilde{\mathbf{q}}^a_b$ are equivalent descriptions.

2.4.1 Angular integration



$\boldsymbol{\omega}^b_{ib}(t_{k+1})$
$\bar{\boldsymbol{\omega}}^b_{ib}(t_k)$
$\boldsymbol{\omega}^b_{ib}(t_k)$
t_k
t_{k+1}

Figure 2.5: Integration of angular velocity. The mean is found and transformed to the Earth frame to integrate orientation.

Let a solution $\boldsymbol{\alpha}^b_{ib}$ be given. Clearly $\dot{\boldsymbol{\omega}}^b_{ib} = \boldsymbol{\alpha}^b_{ib}$. A numerical solution is found using Euler's first order method of numerical integration[8, (14.44)]. With a time step Δt, the method offers a global error $\mathbf{E}_{k+1} = \mathcal{O}(\Delta t)$, and the angular velocity state is evolved as

[4]Without the construction being necessary for implementation, the rotation matrix from a quaternion $\mathbf{q} = (q_w, \mathbf{q}_v)^\top$ is[25, p. 14]: $\mathbf{R}\{\mathbf{q}\} = (q_w^2 - \mathbf{q}_v^\top\mathbf{q}_v)\mathbf{I} + 2\mathbf{q}_v\mathbf{q}_v^\top + 2q_w(\mathbf{q}_v)^\times$

$$\boldsymbol{\omega}_{ib}^b(t_{k+1}) = \boldsymbol{\omega}_{ib}^b(t_k) + \boldsymbol{\alpha}_{ib}^b \Delta t$$

However, it is necessary to obtain the angular velocity $\boldsymbol{\omega}_{eb}^b$ in the Earth frame in order to integrate the orientation. From [8, (6.269)] we have $\vec{\omega}_{ib} = \vec{\omega}_{ie} + \vec{\omega}_{eb}$. Then, clearly

$$\boldsymbol{\omega}_{eb}^b(t_{k+1}) = \boldsymbol{\omega}_{ib}^b(t_{k+1}) - \mathbf{R}_e^b \boldsymbol{\omega}_{ie}^e$$

We also define the mean

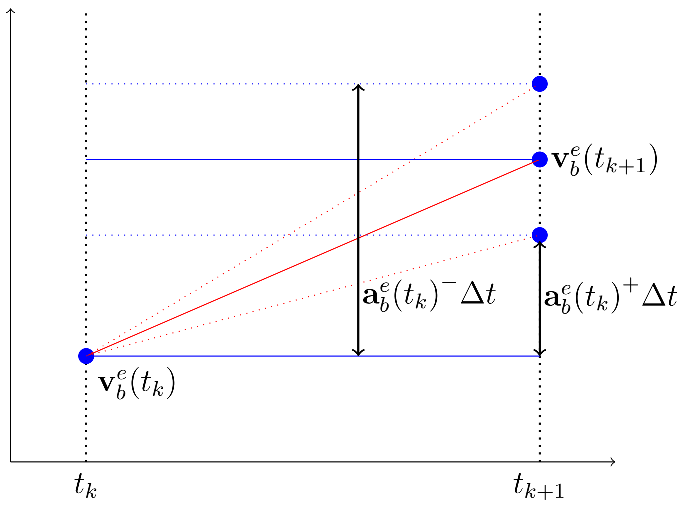
$$\bar{\boldsymbol{\omega}}_{eb}^b(t_k) = \frac{\boldsymbol{\omega}_{eb}^b(t_{k+1}) + \boldsymbol{\omega}_{eb}^b(t_k)}{2}$$

Then from [25, (4.2.2)], a first order angular integration of the orientation in Earth is

$$\mathbf{q}_b^e(t_{k+1}) = \mathbf{q}_b^e(t_k) \otimes \left[\mathbf{q}\left\{ \bar{\boldsymbol{\omega}}_{eb}^b(t_k) \Delta t \right\} + \frac{\Delta t^2}{24} \begin{pmatrix} 0 \\ \boldsymbol{\omega}_{eb}^b(t_k) \times \boldsymbol{\omega}_{eb}^b(t_{k+1}) \end{pmatrix} \right]$$

where[25, (107)] $\mathbf{q}\left\{\phi \mathbf{u}\right\} = \begin{pmatrix} \cos(\phi/2) \\ \mathbf{u}\sin(\phi/2) \end{pmatrix}$ for unit vector $\mathbf{u}$.

2.4.2 Linear integration



$\mathbf{v}_b^e(t_{k+1})$
$\mathbf{a}_b^e(t_k)^- \Delta t$
$\mathbf{a}_b^e(t_k)^+ \Delta t$
$\mathbf{v}_b^e(t_k)$
t_k
t_{k+1}

Figure 2.6: Integration of velocity based on the mean of the acceleration resolved with in the current and next orientation.

Given $\mathbf{a}^e$ from (2.10). Observe that $\mathbf{a}^e$ depends on the orientation $\mathbf{R}_b^e$. Therefore, the acceleration resolved in e will not be same before and after the angular acceleration step. Defining $\mathbf{a}_b^e(t_k)^-$ as the acceleration resolved according to $\mathbf{R}_b^e(t_k)$ before the angular integration, and $\mathbf{a}_b^e(t_k)^+$ resolved according to $\mathbf{R}_b^e(t_{k+1})$ after the integration, we calculate the mean value $\bar{\mathbf{a}}_{eb}^e$ as shown in figure 2.6

$$\bar{\mathbf{a}}^e_{eb}(t_k) = \frac{\mathbf{a}^e_b(t_k)^- + \mathbf{a}^e_b(t_k)^+}{2}$$

Clearly $\dot{\mathbf{v}}^e_b = \mathbf{a}^e_b$, and $\dot{\mathbf{p}}^e_b = \mathbf{v}^e_b$. Employing a semi-implicit variation of Euler's method[38], velocity is first integrated and the updated value used immediately to evolve position:

$$\mathbf{v}^e_b(t_{k+1}) = \mathbf{v}^e_b(t_k) + \bar{\mathbf{a}}^e_{eb}\Delta t$$

$$\mathbf{p}^e_b(t_{k+1}) = \mathbf{p}^e_b(t_k) + \mathbf{v}^e_b(t_{k+1})\Delta t$$

3 Quadrotor Actuation

The equations of motion derived in the last chapter pertains generally to rigid bodies and is applicable for all mobile robots given a set of force and torque actuation inputs. Here we will focus on developing an actuation model for a *quadrotor*, also known as a *quadcopter*. We start with a simple model of a symmetric rigid body with four rotors generating independent thrust and torque and derive the full actuation map for this simplified model. The focus is solely on control actuators, and any potential interfering forces from the environment, like ground collision forces or wind disturbance, have been ignored.

3.1 Forces and torques

Consider a force $\mathbf{f}_j^b$ resolved in body frame, with some line of action through the body. Let $\mathbf{r}_j^b$ be the arm from the mass center c to some arbitrary point on the line of action. Consider also a torque $\boldsymbol{\tau}_j^b$. A *torque* is defined here as the moment of a set of forces with zero resultant force, meaning the moment - or torque, is the same about any point.

For a set S of n_f forces and n_τ torques acting on the body we have the resultant force without specification to a line of action[8, (7.1)]

$$\mathbf{f}_b^b = \sum_{j=1}^{n_f} \mathbf{f}_j^b$$

and the total moment about the center of mass c[8, (7.4)]

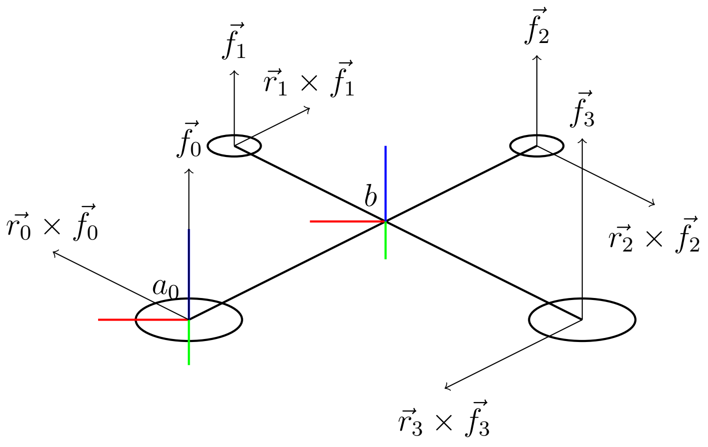
$$\boldsymbol{\mu}_{b/c}^b = \sum_{j=1}^{n_\tau} \boldsymbol{\tau}_j^b + \sum_{j=1}^{n_f} \mathbf{r}_j^b \times \mathbf{f}_j^b \tag{3.1}$$

It can be shown[8, (7.10)] that the set S of forces and torques with resultant force $\mathbf{f}_b^b$ and moment $\boldsymbol{\mu}_{b/c}^b$ about c is equivalent to a force $\mathbf{f}_{bc}^b = \mathbf{f}_b^b$ with a line of action through c in combination with a torque $\boldsymbol{\tau}_{bc}^b = \boldsymbol{\mu}_{b/c}^b$. That is, both sets will have the same resultant force and same momentum about any point on b. For an offset $\mathbf{r}^b$ from c to a point o then from [8, (7.17),(7.18)] we have

$$\begin{pmatrix} \mathbf{f}^b_{bo} \\ \boldsymbol{\tau}^b_{bo} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ (\mathbf{r}^b)^\times & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{f}^b_{bc} \\ \boldsymbol{\tau}^b_{bc} \end{pmatrix}$$

3.2 Quadrotor model

The rigid body quadrotor model will be defined as a symmetric configuration with a set of four rotors with moment arms $\mathbf{r}^b_j$ from the center actuating a lift force and momentum as shown in figure 3.1 as well as individual torques shown in figure 3.2.



$\vec{f}_1$
$\vec{r}_1 \times \vec{f}_1$
$\vec{f}_2$
$\vec{f}_3$
$\vec{f}_0$
b
$\vec{r}_0 \times \vec{f}_0$
$\vec{r}_2 \times \vec{f}_2$
a_0
$\vec{r}_3 \times \vec{f}_3$

Figure 3.1: Quadrotor lift forces and resulting moments about body center.

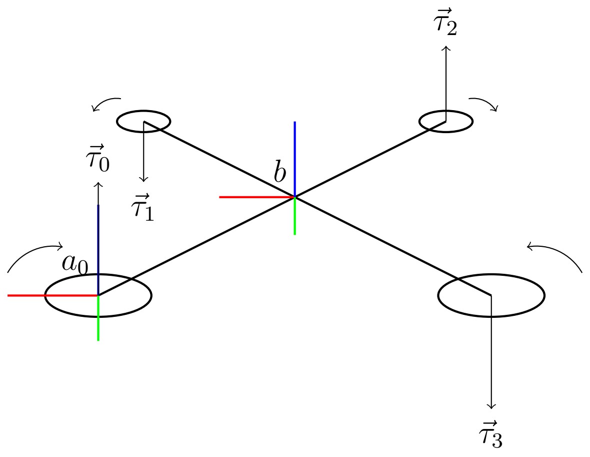


Figure 3.2: Torques generated by the four spinning rotor actuators.

For arms in the symmetric X-configuration numbered $0, .., 3$ as specified in figure 3.1 and body frame as shown, the positions of the actuators are

$$\mathbf{r}_0^b = \begin{pmatrix} \frac{\sqrt{2}}{2}r \\ \frac{\sqrt{2}}{2}r \\ 0 \end{pmatrix}, \ \mathbf{r}_1^b = \begin{pmatrix} \frac{\sqrt{2}}{2}r \\ -\frac{\sqrt{2}}{2}r \\ 0 \end{pmatrix}, \ \mathbf{r}_2^b = \begin{pmatrix} -\frac{\sqrt{2}}{2}r \\ \frac{\sqrt{2}}{2}r \\ 0 \end{pmatrix}, \ \mathbf{r}_3^b = \begin{pmatrix} -\frac{\sqrt{2}}{2}r \\ -\frac{\sqrt{2}}{2}r \\ 0 \end{pmatrix}$$

for a given arm length r. From the actuator frames $\mathbf{a}_j$ defined as in 3.1, 3.2 it is clear that

$$\mathbf{f}_j^{a_j} = \begin{pmatrix} 0 \\ 0 \\ f_z^j \end{pmatrix}, \quad \boldsymbol{\tau}_j^{a_j} = \begin{pmatrix} 0 \\ 0 \\ \tau_z \end{pmatrix}$$

This means it is very convenient to define

$$\mathbf{R}_{a_i}^b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

for all actuators, such that $\mathbf{f}_j^b = \mathbf{f}_j^{a_j}$ and $\boldsymbol{\tau}_j^b = \boldsymbol{\tau}_j^{a_j}$. For reasons of simplicity, and because the symmetric nature of the quadrotor model warrants it, we will assume the body frame is centered in the rigid body mass center. A body frame defined as in 3.1, with axes coinciding with the principal axes of inertia, the axis of symmetry, of the body then the inertia tensor[8, (7.82)] $\mathbf{M}_{bc}^b = \int_b [(\mathbf{r}^b)^2\mathbf{I} - \mathbf{r}^b(\mathbf{r})^\top]dm$ takes on a simple form with the principal moments of inertia on the diagonal:

$$\mathbf{M}_{b/c}^b = \begin{pmatrix} M_{xx} & 0 & 0 \\ 0 & M_{yy} & 0 \\ 0 & 0 & M_{zz} \end{pmatrix}$$

where $M_{xx} = M_{yy}$.

3.3 Rotor model

An actuator setup for a quadrotor will typically feature a high-bandwidth electronic speed controller taking control inputs and applying a voltage duty cycle to drive a brush-less DC motor which torques a driveshaft, spinning the propeller. We will derive a simplified model for the total response, from control input to force output, modeling the dynamics of the components involved, in turn.

3.3.1 Propeller aerodynamics

The airfoil of a spinning propeller blade will experience a lift force L and drag force D respectively normal to and in the direction of the flow across the foil[36]

$$
\begin{aligned}
L &= \frac{1}{2} C_L \rho V^2 A \\
D &= \frac{1}{2} C_D \rho V^2 A
\end{aligned}
$$

where ρ is the density of air, V the speed of the air across the foil, A the area of the foil, and the dimensionless coefficients C_L and C_D are a function of the angle of attack of the foil to the air flow, and the Reynolds number. A combined effect of the lift and drag force creates a thrust T and a torque Q aligned with the the axis of rotation for the propeller[14]

$$
\begin{aligned}
F &= C_F \rho D^4 n^2 \\
Q &= C_Q \rho D^5 n^2
\end{aligned}
$$

where D is the diameter of the propeller, n is the propeller rotations per second, and C_F and C_Q dimensionless coefficients dependent on various geometric propeller properties, the Reynolds number and a dimensionless expression of the propeller's speed of advance. In an effort to simplify modeling with a minimum realization of parameters, we let $F = K_F \omega^2$, $Q = K_Q F$ where the necessary parameters will detailed with the correct units, and values given according to the implementation in section 7.2.

In addition to the drag-induced torque Q at stationary operation, when the propeller rotation speed is changed, the change in angular momentum of the propeller, $\tau = I\dot{\omega}$ after Euler's law, will induce a counter-acting torque $-\tau$ on the body due to conservation of angular momentum. We then have

$$
\mathbf{f}_j^b = \begin{pmatrix} 0 \\ 0 \\ F \end{pmatrix}, \quad \boldsymbol{\tau}_j^b = \begin{pmatrix} 0 \\ 0 \\ \pm(I\dot{\omega} + K_Q F) \end{pmatrix} \tag{3.2}
$$

where $+/-$ accompanies clock-wise and counter-clockwise rotor operation around the body z-axis, respectively.

3.3.2 DC motor electromechanics

An electric motor with rotary motion has a stationary part called the *stator* and a rotary part called the *rotor*. An *armature* current interferes with a permanent electromagnetic *field*, causing relative rotation between the stator and rotor. The motor shaft fixed to the rotor then rotates in response to this internal motor torque. In a constant field DC motor, the armature circuit is attached to the rotor and is driven by a current i_a alternating on/off in response to the applied voltage u_a. A simple model for the dynamics of the motor response is[2, (2.63)-(2.67)]

$$
\begin{aligned}
I\dot{\omega} &= \tau - \tau_e \\
\tau &= K_t i_a \\
e_a &= K_e \omega \\
u_a &= R_a i_a + L_a + \frac{d}{dt} i_a + e_a
\end{aligned}
$$

where ω is the achieved angular shaft speed, I the combined rotor and load moment of inertia, τ the driving torque, τ_e an external load torque, K_t and K_e motor coefficients, e_a the counter-electromotive force, and R_a, L_a parameters of the internal armature circuit. This is concicely modeled as the transfer function[8, (3.52)]

$$
h(s) = \frac{\omega}{u_a}(s) = \frac{\frac{1}{K_e}}{(1 + sT_m)(1 + sT_a)} \tag{3.3}
$$

where $T_a = \frac{L_a}{R_a}$, $T_m = \frac{IR_a}{K_e K_t}$ are the electrical and mechanical time constants, respectively. A reasonable modeling assumption is $L_a << R_a$ such that the dynamic response of the electronics is much faster than the mechanics, giving $T_a << T_m$, and the response time of the electronic circuit can reasonably be neglected from the modeling dynamics. The result is a simple first order impulse response

$$
h(s) = \frac{\frac{1}{K_e}}{1 + sT_m}
$$

3.3.3 Electronic speed controller dynamics

A quadrotor setup will typically feature a brush-less DC (BLDC) motor. Compared to a brushed DC motor, a BLDC motor will have the armature circuit on the stator to avoid having a mechanical interface to feed the armature. BLDC motors are therefore mechanically simple, but require complex control electronics and regulated power supply to sequentially energize the stator coils generating a rotating electric field which drags the rotor around with it[39].

For a given operating range with constant motor load a BLDC motor will have a linear relationship between applied voltage and motor speed, given that the supply voltage is well regulated. This enables the motor to operate synchronously in an open loop fashion, to some extent. However, when load is proportional to shaft speed and the range of operation is substantial, as is the case with quadrotors, a BLDC motor driven in open-loop mode will exhibit severe non-linear characteristics[20] not present in the simple model (3.3).

It is therefore crucial to employ feedback in the electronic speed controller to regulate the speed of the motor. Feedback is available either through Hall-effect sensors embedded in the stator to indicate the relative positions of stator and rotor, or by monitoring the back-induced emf in the windings in a sensorless setup. An electronic feedback speed controller is typically implemented as a PI controller[27].

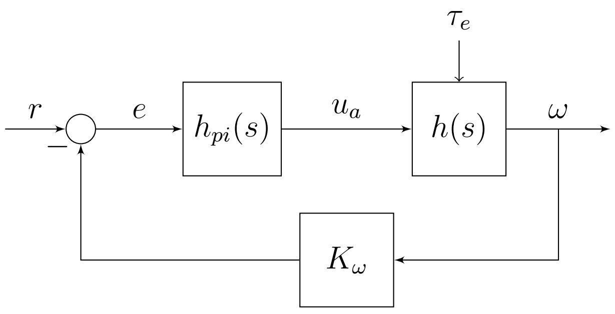


Figure 3.3: Speed feedback control of brush-less DC motor.

The transfer function of a PI controller is given by[2, (9.46)]

$$h_{pi}(s) = \frac{u_a}{r}(s) = K_p \frac{1 + sT_i}{sT_i}$$

for controller gain K_p and time constant T_i. We assume input $r \in [0, 1]$, adjusted for with the angular rate scaling factor K_ω. The closed loop response as seen in figure 3.3 is

$$H(s) = \frac{\omega}{r}(s) = \frac{h_0(s)}{1 + K_\omega h_0(s)}$$

Substituting in $h_0(s) = h_{pi}(s)h(s)$ the result is closed loop impulse response

$$H(s) = \frac{K_p(1 + sT_i)}{sK_eT_i(1 + sT_m) + K_\omega K_p(1 + sT_i)}$$

In principle, the electronic speed controller is a black box, and neither the controller algorithm nor the parameter tuning is known. However, the important characteristics to model are: i) achieving an overall linear response from control input to motor speed output. This is achieved when we employ a linear model both for the motor and the controller. In practice there might be non-linearities in both parts that we do not model, but which in turn by design cancel each other out. ii) The dynamic response of the closed loop system. If we assume a *well-tuned* controller, a reasonable modeling simplification is to assume $T_i \approx T_m$. Canceling terms in $H(s)$ this then allows us to approximate the entire controller and motor dynamic response with a simple first order transfer function

$$H(s) = \frac{K}{1 + sT} \tag{3.4}$$

where $K = \frac{1}{K_\omega}$, $T = \frac{K_eT_i}{K_\omega K_p}$. The corresponding state space model is given by

$$\dot{\omega}(t) = \frac{Kr(t) - \omega(t)}{T}$$

Solving with first order Euler integration we get the motor shaft speed necessary to employ the propeller thrust and torque model in (3.2)

$$\omega(t_{k+1}) = \omega(t_k) + \frac{Kr(t_k) - \omega(t_k)}{T}\Delta t \tag{3.5}$$

All the parameters necessary to implemented this rotor model are specified in section 7.2.

4 Inertial Measurements

An inertial measurement unit is an electronic device measuring proper acceleration and angular velocity using a combination of accelerometers and gyroscopes. Given the initial position, velocity and attitude of the IMU, *dead reckoning* of the current state is possible to achieve through integration. However, due to measurement errors this solution is in practice accurate only for a very limited amount of time and inertial measurements alone are rarely used for navigational purposes without other aiding sensors. Basically, there are two categories of these measurement errors, stochastic and systematic[15]. Systematic errors include scale factor errors, cross-axis sensitivity due to mounting misalignment and g-dependency. Systematic errors are fixed and/or a function of the sensor input and environmental conditions, but can be measured and somewhat compensated for during factory or user calibration, although might degrade over time and during different operating conditions.

Stochastic errors are random in nature, and include white noise disturbances which cannot be removed during calibration. The objective here is to model the output from an IMU in order to simulate its sensor data with the same general characteristics as seen from the point of view of a user. The purpose is then not to model systematic errors which can somewhat be compensated for with calibration, but to add stochastic noise to the inertial signals . This will include discrete white measurement noise and a slowly changing bias.

4.1 Accelerometer

An accelerometer behaves conceptually as a mass spring damper which, when undergoing acceleration, is displaced relative to the surrounding casing. It is important to note an accelerometer measures only proper acceleration[37]. That is, as described in section 2.2.1, acceleration relative to any inertial frame of reference as a direct result of specific non-gravitational forces acting on the body.

Modern accelerometers, often implemented as microelectromechanical systems (MEMS), utilize piezoresistive or capacitive electronic components to convert mechanical motion to an electrical signal. Most such MEMS accelerometers are by design only sensitive to acceleration in one direction. Three such devices can be integrated perpendicularly in-plane and out-of-plane on a single die to offer full three-axis coverage.

4.1.1 Measurement model

For each axis, our measurement model[21, (2)] will include discrete sampling white noise n_a and a slowly changing b_a added to the proper acceleration a such that for the x-axis measurement

$$\begin{aligned} \tilde{a}_x &= a_x + b_{a_x} + n_a \\ n_a &\sim \mathcal{N}(0, \sigma_a^2) \end{aligned} \tag{4.1}$$

where n_a is assumed to be normally distributed with zero mean and σ_a standard deviation. The measurements for the y and z axes are modeled exactly the same way. The bias is modeled as a first order Gauss-Markov process, widely used to model random errors[23, (1)]

$$\begin{aligned} \dot{b}_{a_x} &= -\frac{1}{T_{b_a}} b_{a_x} + n_{b_a} \\ n_{b_a} &\sim \mathcal{N}\left(0, \sigma_{b_a}^2\right) \end{aligned}$$

where n_{b_a} is a continuous white noise driving signal with σ_{b_a} standard deviation. T_{b_a} is the process time constant.

4.1.2 Generating a measurement

Let proper body-accelerations $\mathbf{a}_o^b$, $\boldsymbol{\alpha}_{ib}^b$ be given from solving (2.9). Assume an offset $\mathbf{r}_s^b$ from the body frame to the origin of a user defined sensor frame s presumably located somewhere in the IMU device and, here by definition, aligned with the IMU axes of measurement. Then, the kinematic transform (2.5) gives the proper acceleration of the sensor frame origin, resolved in the body frame according to

$$\mathbf{a}_s^b = \mathbf{a}_o^b + (\boldsymbol{\alpha}_{ib}^b)^\times \mathbf{r}_s^b + (\boldsymbol{\omega}_{ib}^b)^\times (\boldsymbol{\omega}_{ib}^b)^\times \mathbf{r}_s^b$$

Given the orientation of the sensor frame in the body frame, $\mathbf{R}_s^b$, the result in the sensor frame is easily obtained as $\mathbf{a}_s^s = \left(\mathbf{R}_s^b\right)^\top \mathbf{a}_s^b$. Similarly for the angular acceleration, $\boldsymbol{\alpha}_{is}^s = \left(\mathbf{R}_s^b\right)^\top \boldsymbol{\alpha}_{ib}^b$. Assume the placement of each accelerometer axis in the IMU is offset a small distance from the chosen sensor frame origin. Let $\mathbf{r}_x^s$ be the offset from the origin to the x-axis accelerometer. Then again from (2.5) we have

$$\mathbf{a}_x^s = \mathbf{a}_s^s + (\boldsymbol{\alpha}_{is}^s)^\times \mathbf{r}_x^s + (\boldsymbol{\omega}_{is}^s)^\times (\boldsymbol{\omega}_{is}^s)^\times \mathbf{r}_x^s$$

Notice that with $\mathbf{r}_x^s = 0$, $\mathbf{a}_x^s = \mathbf{a}_s^s$. The single-axis acceleration a_x is then the element corresponding to the axis: $a_x = \mathbf{a}_x^s(0)$. Similary we find $a_y = \mathbf{a}_y^s(1)$ and $a_z = \mathbf{a}_z^s(2)$ constructed with offsets $\mathbf{r}_y^s$, $\mathbf{r}_z^s$.

Adding noise and bias according to the measurement model (4.1), we get the discrete sampling of the measurement signal;

$$\tilde{a}_x(t_k) = a_x(t_k) + b_{a_x}(t_k) + n_a(t_k)$$

The bias is discretely updated according to[23, (2)]

$$b_{a_x}(t_k) = \left(1 - \frac{\Delta t}{T_{b_a}}\right) b_{a_x}(t_k) + n_{b_a}(t_k)\sqrt{\Delta t}$$

for a sampling period Δt. The process to generate y and z axis measurements is identical. How to generate n_a^k and $n_{b_a}^k$ and initialize the bias process, is detailed in the *Implementation* of section 7.3. After clamping the results to accommodate for accelerometer range limitations and clipping, the combined measurements give a 3-axis measurement in the sensor frame:

$$\tilde{\mathbf{a}}_s^s(t_k) = (\tilde{a}_x(t_k), \tilde{a}_y(t_k), \tilde{a}_z(t_k))^\top$$

4.2 Gyroscope

A modern gyroscope is typically a vibrating structure gyroscope implemented with MEMS technology. The construction is simpler than conventional rotating disc gyroscopes, and operate by determining the rate of rotation from the Coriolis force acting on the supporting structure of a vibrating object when undergoing rotation. The gyroscope measures angular velocity around one axis, and similarly to the accelerometer three perpendicular gyroscopes can be arranged to form a 3-axis measurement.

4.2.1 Measurement model

Using the identical noise and bias measurement model from (4.1), the angular velocity around the x axis is

$$\begin{aligned} \tilde{\omega}_x &= \omega_x + b_{\omega_x} + n_\omega \\ n_\omega &\sim \mathcal{N}(0, \sigma_\omega^2) \end{aligned}$$

and equivalent for the y and z axis. The discrete sampling noise n_ω is assumed to be normally distributed with zero mean and σ_ω standard deviation. Again the bias is modeled as a first order Gauss-Markov process

$$
\begin{aligned}
\dot{b}_{\omega_x} &= -\frac{1}{T_{b_\omega}} b_{\omega_a} + n_{b_\omega} \\
n_{b_\omega} &\sim \mathcal{N}\left(0, \sigma_{b_\omega}^2\right)
\end{aligned}
$$

where n_{b_ω} is a continuous white noise driving signal with a σ_{b_ω} standard deviation. T_{b_ω} is the time constant.

4.2.2 Generating a measurement

Let $\boldsymbol{\omega}_{ib}^b$ be given from the rigid body state vector. Given $\mathbf{R}_s^b$ then $\boldsymbol{\omega}_{is}^s = (\mathbf{R}_s^b)^\top \boldsymbol{\omega}_{ib}^b$ for the same IMU sensor frame provided for the accelerometer. We let the axes of the sensor frame determine the axes of measurement, so by definition they are both aligned. However, compared to the accelerometer, it is irrelevant where the sensor frame is located relative to the gyroscopes as long as the axes align - the angular velocity will be the same. Therefore there is no need to decompose $\boldsymbol{\omega}_{is}^s$ further, and the 3-axis generated measurement is simply expressed as

$$
\tilde{\boldsymbol{\omega}}_{is}^s(t_k) = \boldsymbol{\omega}_{is}^s(t_k) + \boldsymbol{b}_\omega(t_k) + \boldsymbol{n}_\omega(t_k)
$$

where the bias is updated according to

$$
\boldsymbol{b}_\omega(t_k) = \left(1 - \frac{\Delta t}{T_{b_\omega}}\right) \boldsymbol{b}_\omega(t_k) + \boldsymbol{n}_{b_\omega}(t_k)\sqrt{\Delta t}
$$

How to generate $\boldsymbol{n}_\omega^k$ and $\boldsymbol{n}_{b_\omega}^k$ and initialize the Gauss-Markov process will be detailed in section 7.3.

5 Camera Images

A camera is an optical instrument for capturing images of a scene onto photographic film, or an electronic image sensor in the case of a modern digital camera. The camera lense captures the incoming light and brings it to focus on the image sensor where light is turned into discrete signals. Camera images can provide a mobile robot with valuable information on scene structure and content. Distinct and recognizable feature patches can be tracked from image to image, and the process of triangulation can be used to estimate the position of a feature in three-dimensional space to infer the motion of the robot relative to the scene and/or scene depth.

In this chapter we will discuss the generation of camera images from a given virtual 3D scene. The perspective properties of a specific camera will be emulated according to its intrinsic camera calibration parameters. In addition a considerable effort is made to simulate the effects of optical lense distortion to generate images that can be input to the rest of the system as a drop-in replacement for real images exhibiting these distortion effects.

5.1 Pinhole camera

The word camera is derived from the latin *camera obscura*, a natural optical phenomenon that occurs when an image of a scene is projected onto a flat surface in a dark room through a small hole allowing light in from the outside. The image that forms on the surface is an inverted (left-right and top-bottom) two-dimensional perspective projection of the scene. An pinhole camera is the most straightforward implementation of the camera obscura effect; a light-proof box with a tiny aperture pinhole without a lens projecting an inverted image onto the opposite side of the box.

5.1.1 Pinhole camera model

The pinhole camera model[35] describes the perspective transformation of an ideal pinhole camera, from a point $\mathbf{x}^c = (X, Y, Z)^\top$ in the camera frame to a point $(x, y)^\top$ in the image plane a focal length f' away from the pinhole, as shown in figure 5.1

$$\begin{pmatrix} x \\ y \end{pmatrix} = -\frac{f'}{Z} \begin{pmatrix} X \\ Y \end{pmatrix}$$

Notice that the perspective transformation scales with the focal length f', the distance from the aperture to the image plane. Also notice the negative relation which as evident from figure 5.1 inverts coordinates from left to right and top to bottom on the image.

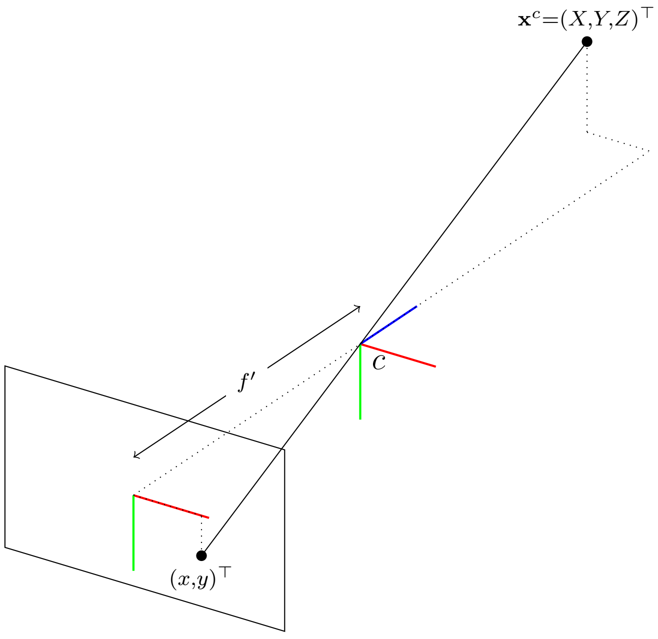


Figure 5.1: Pinhole camera projection. A point $\mathbf{x}^c$ in the camera frame, centered at the pinhole aperture, is projected onto a flat image plane a focal length f' away from the pinhole.

5.1.2 Pixel coordinate transform

The pinhole camera model is extended to map the image coordinates to pixel coordinates, to facilitate the pixelization of a discrete image sensor. Let d_x and d_y be the size of a pixel in metric units in the horizontal and vertical direction of the image plane, respectively, to allow for a non-square image sensor. We want the top left corner of an image to have pixel coordinate $(0,0)^\top$. Defining the pixel coordinate of the center point (c_x, c_y), the metric units to pixels coordinate mapping is then given by

$$(x,y)^\top \mapsto \left(\frac{x}{d_x} + c_x, \frac{y}{d_y} + c_y\right)^\top$$

Additionally, to simplify the pinhole camera model to do away with the inverted projection, we project onto a virtual image plane as if in front of the pinhole. Let the virtual image plane be placed the same metric focal length f' in front of the pinhole as shown in figure 5.2. For simplicity we define the focal lengths $f_x = \frac{f'}{d_x}$, $f_y = \frac{f'}{d_y}$ in pixel units. The pinhole projection model to pixel coordinates is then

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c_x \\ c_y \end{pmatrix} + \begin{pmatrix} f_x & 0 \\ 0 & f_y \end{pmatrix} \begin{pmatrix} \frac{X}{Z} \\ \frac{Y}{Z} \end{pmatrix} \tag{5.1}$$

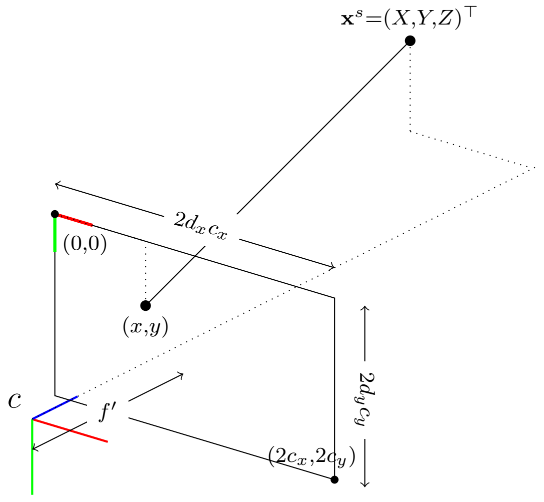


Figure 5.2: Pinhole camera projection to pixel coordinates. A point $\mathbf{x}^c$ is projected onto a virtual image plane a metric focal length f' in front of the center of the camera frame.

The parameters f_x, f_y, c_x and c_y that make up the perspective projection are called the *intrinsic parameters* of the camera.

5.2 OpenGL camera

Open Graphics Library (OpenGL) is a cross-language computer graphics API to achieve GPU accelerated rendering of 2D and 3D vector graphics. It is cross-platform and offers numerous language bindings to be callable from most popular programming languages[30]. We will use the concept of an OpenGL camera to render a 3D scene to a 2D image, simulating the perspective properties of a given set of intrinsic parameters.

5.2.1 Viewing Frustum

In computer graphics the viewing frustum determines the region of space in front of a camera that will be viewable and rendered to a screen[1]. Objects not inside the frustum, for example in front of the near plane or behind the far plane, will be clipped from view and not rendered. The shape of the frustum, that is the width and height in relation to the near plane distance from the camera, determines the perspective projection of the

[1]Or a non-default user created FrameBuffer Object for off-screen rendering.

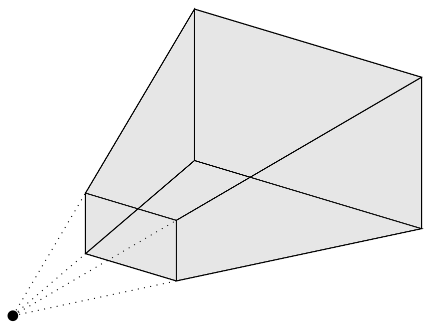


Figure 5.3: The viewing frustum determines the region of space in front of a camera that will be rendered.

camera. This means the viewing frustum is also a perspective frustum. The frustum parameters are then used to construct the OpenGL projection matrix.

5.2.2 Projection matrix

We will make a difference between perspective *projection* and perspective *transformation*. A perspective projection is performed by the pinhole camera model, where the X and Y coordinates are divided through by Z to achieve perspective scaling, and the Z value discarded as the projection is onto a flat image surface always a focal length away from the pinhole. In computer graphics, a perspective transformation is used to map $(X, Y, Z) \mapsto (x, y, z)$ where X and Y are scaled perspectively but a notion of the Z value is kept in z as the depth in the scene relative to the near and far planes. The z-buffer value is in turn used to perform depth-testing and back-face culling to determine what objects should be rendered or are culled from view behind others.

To achieve this, *homogenous coordinates*[33] are used. For a 3D projective space, 3+1 homogeneous coordinates are needed to represent a point, where an extra coordinate W is added. $W = 0$ is used to represent points at infinity using finite coordinates, and is otherwise a scaling factor to achieve perspective forshorterning the same way the Z value is used in perspective projection. This then allows perspective transformations to be mathematically possible and expressable as a matrix multiplication. Perhaps confusingly, in OpenGL this matrix is called the *Projection matrix*

Consider the point $\mathbf{x}^{c_g} = (X, Y, Z)^\top$ in an OpenGL camera frame c_g as defined in figure 5.4. Notice the camera is looking down its negative z-axis. In homogeneous coordinates the same point is represented as $(\mathbf{x}^{c_g}, W)^\top = (X, Y, Z, W)^\top$, where W is the scaling factor and $W = 1$ for *normal* perspective scaling. For the viewing frustum parameters l, f, b, t, n, f_z[2] as shown in figure 5.4, the OpenGL projection matrix[4, (II.18)]

[2]Not to be confused with the focal length f

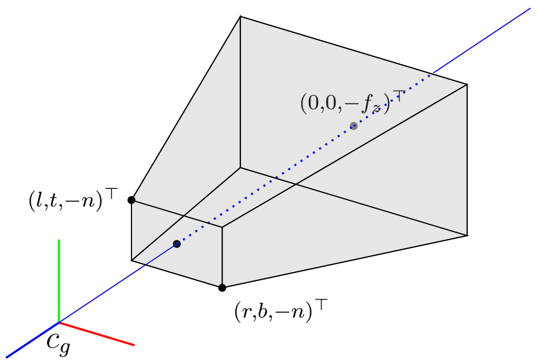


Figure 5.4: OpenGL perspective frustum and camera frame c_g. The shape of the frustum defines the properties of the perspective transformation and its parameters are used to define the projection matrix.

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f_z+n)}{f_z-n} & \frac{-2f_zn}{f_z-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} \tag{5.2}$$

defines the perspective transform from homogeneous coordinates to the clip coordinates $(x_c, y_c, z_c, w_c)^\top$. Notice that despite the near and far plane being down the negative z-axis from the camera, the frustum parameters n and f_z are actually positive values, due to an OpenGL convention. The actual perspective forshortening is achieved by diving through the $w_c = -Z$ coordinate, where the scaling parameter w_c by default is set to 1 and discarded, yielding the normalized coordinates

$$\begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}$$

If any coordinate is is outside the range $-w_c < x_c, y_c, z_c < w_c$ it is outside the viewing frustum and clipped from view, so the resulting coordinates are normalized $-1 < x_n, y_n, z_n < 1$, where the projection matrix actually maps $-z_n$ to $+1$ and so forms a left-handed coordinate system.

5.2.3 View matrix

The projection matrix transforms coordinates defined in the camera space, $\mathbf{x}^{c_g} = (X, Y, Z)^\top$. In world coordinates the same point is given by $\mathbf{x}_p^w = \mathbf{r}_c^w + \mathbf{R}_{c_g}^w \mathbf{x}^{c_g}$ where $\mathbf{r}_c^w$ is the position of the camera in the world frame, resolved in the world frame. Conversely, to express a point $\mathbf{x}_p^w$ in the camera frame, we have $\mathbf{x}^c = \mathbf{R}_{c_g}^{w\,\top}(\mathbf{x}^{c_g} - \mathbf{r}_c^w)$.

Let the corresponding homogeneous coordinate be $(\mathbf{x}^{c_g}, 1)^\top$ for normal perspective scaling $W = 1$ out of simplicity. Constructing a homogeneous transformation matrix, the perspective transformation(5.2) from world coordinates becomes

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f_z+n)}{f_z-n} & \frac{-2f_z n}{f_z-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}}_{\text{Projection matrix}} \underbrace{\begin{pmatrix} \mathbf{R}_{c_g}^{w\,\top} & -\mathbf{R}_{c_g}^{w\,\top}\mathbf{r}_c^w \\ \mathbf{0}^\top & 1 \end{pmatrix}}_{\text{View matrix}} \begin{pmatrix} \mathbf{x}_p^w \\ 1 \end{pmatrix} \tag{5.3}$$

The *view matrix* then describes the position and orientation of the world coordinate frame, as seen from the camera frame, and is used to appropriately position the camera in relation to world objects.

5.3 Constructing OpenGL camera from pinhole camera

To emulate the camera images from a specific camera, modeled as an ideal pinhole camera, the OpenGL viewing frustum needs to be specified in terms of the intrinsic pinhole camera parameters.

5.3.1 Projection matrix

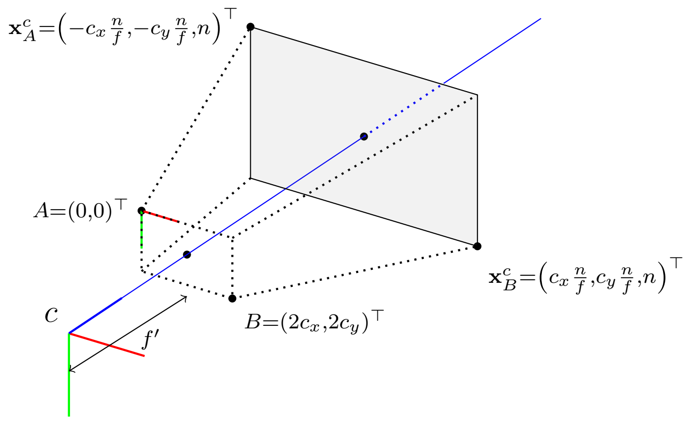

$\mathbf{x}_A^c=\left(-c_x\frac{n}{f},-c_y\frac{n}{f},n\right)^\top$
$A=(0,0)^\top$
$\mathbf{x}_B^c=\left(c_x\frac{n}{f},c_y\frac{n}{f},n\right)^\top$
C
f'
$B=(2c_x,2c_y)^\top$

Figure 5.5: Perspective unprojection, from 2D pixel coordinates to 3D points on the near plane.

Consider an ideal pinhole projection model, with an ideal square pixels image sensor and resulting focal length in pixel units $f = \frac{f'}{d}$, where d is the side length of a pixel in metric units. Then from (5.1)

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c_x \\ c_y \end{pmatrix} + f \begin{pmatrix} \frac{X}{Z} \\ \frac{Y}{Z} \end{pmatrix}$$

The inverse operation, *unprojecting* an image pixel coordinate to a 3D space, is readily given by

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \frac{Z}{f} \begin{pmatrix} x - c_x \\ y - c_y \end{pmatrix} \tag{5.4}$$

for any given Z value. Consider the image coordinates $A = (0,0)^\top$ and $B = (2c_x, 2c_y)^\top$ as shown in figure 5.5. Unprojecting these points to corresponding points $\mathbf{x}_A^c$, $\mathbf{x}_B^c$ on the near plane a distance $Z = n$ away from the camera, leads to

$$\begin{aligned} \mathbf{x}_A^c &= \left(-c_x \frac{n}{f}, -c_y \frac{n}{f}, n\right)^\top \\ \mathbf{x}_B^c &= \left(c_x \frac{n}{f}, c_y \frac{n}{f}, n\right)^\top \end{aligned}$$

defined in the pinhole camera frame c as shown in figure 5.5. From figure 5.4 the same points on the near plane are defined in the OpenGL camera frame c_g, in terms of the viewing frustum parameters, according to

$$\begin{aligned} \mathbf{x}_A^{c_g} &= (l, t, -n)^\top \\ \mathbf{x}_B^{c_g} &= (r, b, -n)^\top . \end{aligned}$$

Comparing the camera frames in the two figures it is clear that a mapping between frames is given by $(X, Y, Z)^\top \mapsto (X, -Y, -Z)^\top$. We then have the relation

$$\begin{aligned} (l, r) &= \left(-c_x \frac{n}{f}, c_x \frac{n}{f}\right) \\ (b, t) &= \left(-c_y \frac{n}{f}, c_y \frac{n}{f}\right) \end{aligned} \tag{5.5}$$

This then determines the shape of the viewing frustum, for a given near plane parameter n. The far plane parameter f_z does not affect the perspective transformation, only the resulting z value which is used for depth-testing and back-face culling. The parameter f_z can in principle be chosen arbitrarily as long as the scene is correctly clipped, but be aware that the distance between the near and far plane should be as short as possible due to a depth precision problem at the far plane causing a potential depth precision error known as *z-fighting*[1].

5.3.2 View matrix

To construct the view matrix let the observed body state $\mathbf{r}_b^w$, $\mathbf{R}_b^w$ be given. Let the camera have a body frame offset $\mathbf{r}_c^b$, $\mathbf{R}_{bc}^b$. Apparent from figures 5.4, 5.5 the orientation of the

OpenGL camera is given statically in the camera frame as $\mathbf{R}^c_{c_g}$, involving a 180° rotation around the x-axis. Then the entries for the view matrix from (5.3) then are

$$\begin{aligned}\mathbf{r}^w_c &= \mathbf{r}^w_b + \mathbf{R}^w_b \mathbf{r}^b_{bc} \\ \mathbf{R}^w_{c_g} &= \mathbf{R}^w_b \mathbf{R}^b_c \mathbf{R}^c_{c_g}\end{aligned}$$

5.4 Optical distortion

Due to the effect of using a lens in an actual camera, the projected image will experience *distortion*, a deviation from rectilinear projection. The main effect is straight lines in a scene not remaining straight in the image. This will be very noticeable with a wide-angle lens.

5.4.1 Distortion models

Consider the pinhole projection model of (5.6). Let $(u,v)^\top = \left(\frac{X}{Z}, \frac{Y}{Z}\right)^\top$, and define $\mathbf{h}_0 : \mathcal{R}^2 \mapsto \mathcal{R}^2$ such that

$$\mathbf{h}_0\left(\begin{pmatrix} u \\ v \end{pmatrix}\right) = \begin{pmatrix} c_x \\ c_y \end{pmatrix} + \begin{pmatrix} f_x & 0 \\ 0 & f_y \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \tag{5.6}$$

To simplify notation we will abuse the vector function notation $(x,y)^\top = \mathbf{f}(u,v)$ to have the same exact meaning as $(x,y)^\top = \mathbf{f}((u,v)^\top)$ within this context. To add lens distortion to the pinhole model, let $\mathbf{f} : \mathcal{R}^2 \mapsto \mathcal{R}^2$ be a distorting function such that the nested function

$$\mathbf{h}(u,v) = \mathbf{h}_0\left(\mathbf{f}(u,v)\right)$$

is the final perspective projection to distorted image coordinates $(x,y)^\top = \mathbf{h}(u,v)$. A commonly used distortion model includes radial distortion as well as tangential distorion[24, (5.4)-(5.7)]

$$\begin{aligned}\mathbf{f}(u,v) &= d_r \begin{pmatrix} u \\ v \end{pmatrix} + \boldsymbol{d_t} \\ d_r &= \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ \boldsymbol{d}_t &= \begin{pmatrix} 2uvt_1 + (r^2 + 2u^2)t_2 \\ 2uvt_2 + (r^2 + 2v^2)t_1 \end{pmatrix}\end{aligned} \tag{5.7}$$

for the radial distance $r^2 = u^2 + v^2$ and distortion parameters k_1, k_2, k_3, t_1, t_2. While this function might be sufficient to model barrel distortion in typical wide-angle lenses, for extra wide-angle lenses the fisheye camera model is more applicable.

5.4.2 Fisheye camera model

A fisheye camera has an ultra wide-angle lens producing strong distortion intended for panoramic or hemispherical images, by forgoing entirely with rectilinear projection. The basic effects of a fisheye lens can be modeled as[18]

$$\mathbf{g}_0(u, v) = \frac{\arctan(r)}{r} \begin{pmatrix} u \\ v \end{pmatrix}$$

such that $(x, y)^\top = \mathbf{h}_0 (\mathbf{g}_0(u, v))$ is an ideal fisheye camera model. Similarly to the pinhole camera model, the fisheye camera model can also be extended with radial and tangential distortion effects similar to (5.7), according to[18]

$$\mathbf{f}(u, v) = \begin{pmatrix} d_\theta & \alpha d_\theta \\ 0 & d_\theta \end{pmatrix} \mathbf{g}_0(u, v) \tag{5.8}$$

$$d_\theta = (1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8)$$

$$\theta = \frac{\arctan(r)}{r} \tag{5.9}$$

for distortion parameters $\alpha, k_1, k_2, k_3, k_4$. For small $\alpha << 1$, it is clear that $\mathbf{g}_0$ is a good approximation for $\mathbf{f}$, where only the dominant radial fisheye effects are modeled.

5.4.3 Image distortion

By applying the distortion function, the lense distortion effects of a real image can be added to an ideal generated image.

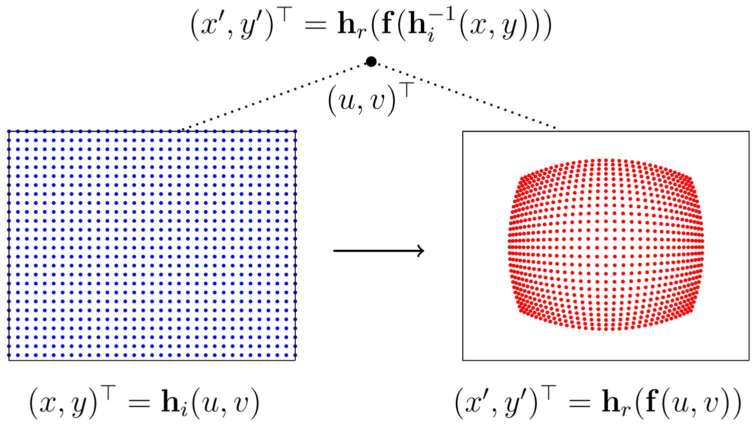

$(x', y')^\top = \mathbf{h}_r(\mathbf{f}(\mathbf{h}_i^{-1}(x, y)))$
$(u, v)^\top$
$(x, y)^\top = \mathbf{h}_i(u, v)$
$(x', y')^\top = \mathbf{h}_r(\mathbf{f}(u, v))$

Figure 5.6: Direct method of image distortion. Each pixel in the undistorted source image (blue) is fed directly to the distorted destination image (red). The mapping is visualized with $\mathbf{f}(u, v) = \mathbf{g_0}(u, v)$.

Direct method

Consider two images constructed with the following projections

$$\begin{aligned}(x, y)^\top &= \mathbf{h}_i(u, v)\\ (x', y')^\top &= \mathbf{h}_r\left(\mathbf{f}(u, v)\right)\end{aligned} \tag{5.10}$$

as shown in figure 5.6. The distortion function is only applied to one projection, and the projection transforms $\mathbf{h}_i$, $\mathbf{h}_r$ are not necessarily the same. Let the images be called the distorted image and the undistorted image, respectively, for the projection with and without the distortion function applied. From (5.10) it is tempting to write the distorted image coordinates in terms of the undistorted:

$$(x', y')^\top = \mathbf{h}_r\left(\mathbf{f}\left(\mathbf{h}_i^{-1}(x, y)\right)\right) \tag{5.11}$$

where the inverse unprojection function is readily available:

$$(u, v)^\top = \mathbf{h}_i^{-1}(x, y) = \begin{pmatrix} \frac{1}{f_x} & 0 \\ 0 & \frac{1}{f_y} \end{pmatrix} \begin{pmatrix} x - c_x \\ y - c_y \end{pmatrix}$$

We then have defined what we will call the *distortion map* $\boldsymbol{d} : (x, y) \mapsto (x', y')$, such that $\boldsymbol{d}(x, y) = \mathbf{h}_r\left(\mathbf{f}\left(\mathbf{h}_i^{-1}(x, y)\right)\right)$. To distort an image it is possible apply the value for each source pixel $(x, y)^\top$ directly to the destination $\boldsymbol{d}(x, y)$ in the distorted image being generated. The distortion map is fine to apply directly on a set of sparse pixels as shown in figure 5.6, but turns out to be problematic in order to distort an image in practice as the direct feed-forward of undistorted source pixels does not guarantee a smooth coverage of all the pixels in the distorted destination image. We will call this the *direct method* of image distortion. In contrast, it turns out there is an alternative way to distort an image using the inverse distortion function.

Indirect method

Going back to (5.10) we instead write the coordinates of the undistorted image in terms of the distorted image:

$$(x, y)^\top = \mathbf{h}_i\left(\mathbf{f}^{-1}\left(\mathbf{h}_r^{-1}(x', y')\right)\right) \tag{5.12}$$

Compared to (5.11), $(u, v)^\top = \mathbf{f}^{-1}\left(\mathbf{h}_r^{-1}(x', y')\right)$ is not readily available due to the inverse distortion function $\mathbf{f}^{-1}$, as the inverse of high-degree polynomial like (5.7) and (5.8) is practically impossible to solve through algebraic manipulation. A solution is found through iterative methods.

For a given coordinate $(x', y')^\top$, the problem is finding $(u, v)^\top = \mathbf{f}^{-1}(u', v')$, where $(u', v')^\top = \mathbf{h}_r^{-1}(x', y')^\top$. Equivalently stated, find a $(u, v)^\top$ such that $(u', v')^\top = \mathbf{f}(u, v)$.

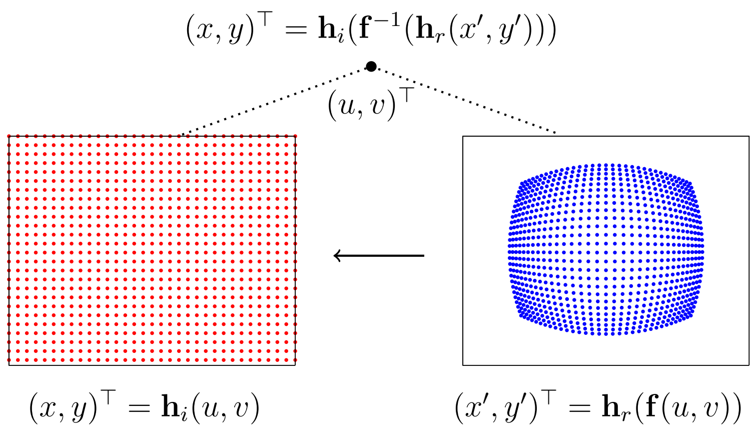

$(x, y)^\top = \mathbf{h}_i(\mathbf{f}^{-1}(\mathbf{h}_r(x', y')))$
$(u, v)^\top$
$(x, y)^\top = \mathbf{h}_i(u, v)$
$(x', y')^\top = \mathbf{h}_r(\mathbf{f}(u, v))$

Figure 5.7: Indirect method of image distortion. Each pixel in the distorted destination image (blue) is pulled from the undistorted source image (red). The mapping is visualized with $\mathbf{f}(u, v) = \mathbf{g_0}(u, v)$.

This allows for an iterative solution searching for the $(u, v)^\top$ that best fits $(u', v')^\top$, using only the direct application of $\mathbf{f}$. We solve this using Gauss-Newton minimization[32], with the residual function

$$\mathbf{r}\left(\begin{pmatrix} u \\ v \end{pmatrix}\right) = \begin{pmatrix} u' \\ v' \end{pmatrix} - \mathbf{f}\left(\begin{pmatrix} u \\ v \end{pmatrix}\right)$$

The iterative steps are then given by

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} = \begin{pmatrix} u_k \\ v_k \end{pmatrix} + \left(\mathbf{J_f}^\top \mathbf{J_f}\right)^{-1} \mathbf{J_f}^\top \mathbf{r}\left(\begin{pmatrix} u_k \\ v_k \end{pmatrix}\right)$$

for the Jacobian $\mathbf{J_f}$. The procedure, including stating the Jacobian for a given distortion function, is given in section 7.4. We then define what we will call the *undistortion map* $\boldsymbol{u} : (x', y') \mapsto (x, y)$, such that $\boldsymbol{u}(x', y') = \mathbf{h}_i\left(\mathbf{f}^{-1}\left(\mathbf{h}_r^{-1}(x', y')\right)\right)$. To distort an image it is clear that for each destination pixel $(x', y')^\top$ in the distorted image, the appropriate source value is located in the undistorted image at pixel coordinate $\boldsymbol{u}(x', y')$. While the iterative process of finding the best inverse function fit for each pixel is time consuming, the entire map is easily precomputable, as the coordinate map is the same regardless of pixel values for each image.

The only remaining problem is discretization of coordinates. While the map inputs discrete pixel coordinates, the result is not necessarily so. Truncating coordinates directly will result in aliasing issues. A solution is found by bilinear interpolation. For every source coordinate $(x, y)^\top = \boldsymbol{u}(x', y')^\top$, let $(i, j)^\top = (\lfloor x \rfloor, \lfloor y \rfloor)^\top$. Let the differences between the actual coordinates and the floored pixel coordinates be

$$\begin{aligned} d_x &= x - i \\ d_y &= y - j \end{aligned}$$

The final distorted image value $v(x', y')$ is then found by bilinear interpolation[31] the source image values at the four corner points

$$v(x', y') = \begin{pmatrix}(1 - d_x) & d_x\end{pmatrix} \begin{pmatrix} v(i, j) & v(i, j + 1) \\ v(i + 1, j) & v(i + 1, j + 1)\end{pmatrix} \begin{pmatrix}(1 - d_y) \\ d_y\end{pmatrix}$$

5.4.4 Image undistortion

A closely related operation is that of image undistortion, undoing the effects of lense distortion. Typically, image undistortion would be used on the receiving end of the camera images, for example in a feature tracker. To simulate the images of a real camera, only image distortion operations like the one already presented is necessary. However, we will still derive the method of image undistortion here as it is helpful to use undistortion to verify the results of a distortion operation. Similarly to image distortion, image undistortion is also readily describable in terms of direct and indirect application of the undistortion map and distortion map, respectively. Of course the direct method here still has the same coverage problem as for image distortion which makes it unsuited for practical application.

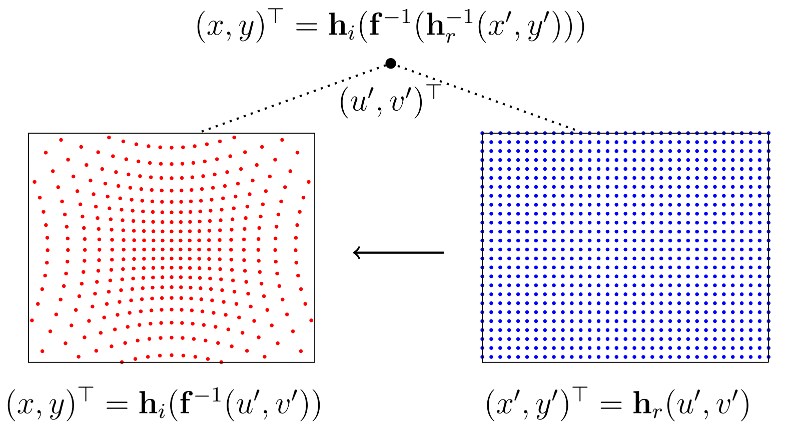

$(x, y)^\top = \mathbf{h}_i(\mathbf{f}^{-1}(\mathbf{h}_r^{-1}(x', y')))$
$(u', v')^\top$
$(x, y)^\top = \mathbf{h}_i(\mathbf{f}^{-1}(u', v'))$
$(x', y')^\top = \mathbf{h}_r(u', v')$

Figure 5.8: Direct method of image undistortion. Each pixel in the distorted source image (blue) is fed directly to the undistorted destination image (red). The mapping is visualized with $\mathbf{f}^{-1}(u, v) = \mathbf{g_0}^{-1}(u, v)$ computed through Gauss-Newton.

Consider again the same exact two image projections from (5.10). To undistort an image, we want to apply a distortion-type operation to the already distorted image. We want to motivate the idea that image undistortion is the *exact* inverse operation of image distortion. To make it clear, we can express the same two image projections, but with an inverse distortion operation applied to the rectilinear, undistorted image, and present the distorted image as a seemingly rectilinear pinhole projection. This is readily shown if we let $(u', v')^\top = \mathbf{f}(u, v)$. Then an equivalent description of (5.10) is

$$\begin{aligned}(x, y)^\top &= \mathbf{h}_i(\mathbf{f}^{-1}(u', v') \\ (x', y')^\top &= \mathbf{h}_r\,(u', v')\end{aligned}$$

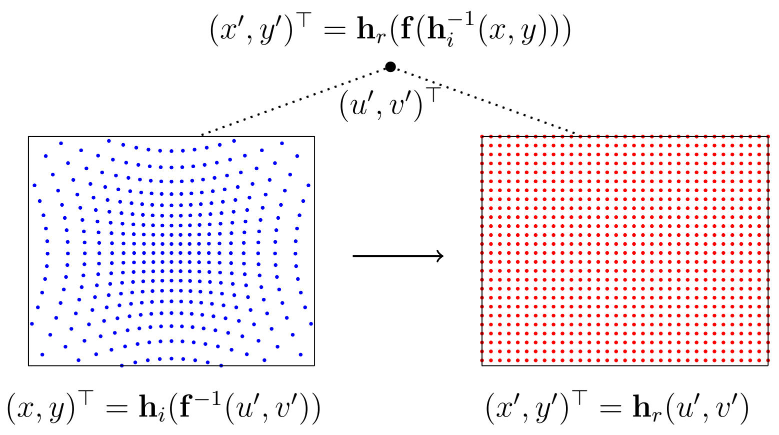

$(x', y')^\top = \mathbf{h}_r(\mathbf{f}(\mathbf{h}_i^{-1}(x, y)))$
$(u', v')^\top$
$(x, y)^\top = \mathbf{h}_i(\mathbf{f}^{-1}(u', v'))$
$(x', y')^\top = \mathbf{h}_r(u', v')$

Figure 5.9: Indirect method of image distortion. Each pixel in the undistorted destination image (blue) is pulled from the distorted source image (red). The mapping is visualized with $\mathbf{f}^{-1}(u, v) = \mathbf{g_0}^{-1}(u, v)$ computed through Gauss-Newton.

Using the same exact distortion map from (5.11) and undistortion map (5.12), figures 5.8, 5.9 show the direct method of image undistortion through applying the undistortion map on the distorted image, and the indirect method through applying the distortion map over the undistorted image. Comparing with image distortion we see that it is in principle arbitrary what we define as the *distorted* image to begin with, as long as the correct distortion map is applied to the correct image in order to achieve the wanted effect. To state it explicitly, we have the direct and indirect methods of image undistortion:

Direct method To undistort an image it is possible to apply the value for each source pixel $(x', y')^\top$ in the distorted image directly to the destination $(x, y)^\top = \boldsymbol{u}(x', y')$ in the undistorted image being generated.

Indirect method To undistort an image it is clear that for each destination pixel $(x, y)^\top$ in the undistorted image, the appropriate source value is located in the distorted image at pixel coordinate $(x', y')^\top = \boldsymbol{d}(x, y)$.

There is an apparent duality between image distortion and undistortion, as presented in figure 5.10. As discussed, direct methods are useless in practice, and indirect methods are to be preferred. A proof by example is given in section 7.4.

	DIRECT	INDIRECT
DISTORTION	$\mathbf{h}_r\mathbf{f}\mathbf{h}_i^{-1}$	$\mathbf{h}_i\mathbf{f}^{-1}\mathbf{h}_r^{-1}$
UNDISTORTION	$\mathbf{h}_i\mathbf{f}^{-1}\mathbf{h}_r^{-1}$	$\mathbf{h}_r\mathbf{f}\mathbf{h}_i^{-1}$

Figure 5.10: Duality of image distortion and undistortion, in terms application of the distortion and undistortion maps.

6 Laser Range Measurements

A Lidar, also known as a *light radar*, or as the acronym LiDAR for *Light Detection And Ranging*, is an active laser sensor scanner which emits laser light pulses, and the time of flight of the return signal is determined to measure the distance travelled. The returning light is not a pure mirror-like reflection, but the result of a *backscattering* effect. The lidar can provide a mobile robot with sampled range measurements of the environment to infer an understanding of scene structure in a way which compliments passive camera imagery. There are two principal ways to model lidar range measurements[22]:

- A depth image sampling across the field of view of the lidar simultaneously, without concern for beam dynamics or the movement of the sensor in between samples.
- A series of individual reflected signals that in turn are emitted, processed and converted to range measurements.

There is a general trade-off between modelling accuracy and run-time performance. The former method is achievable using computer graphics depth buffer rendering and sampling and is in general quite fast. The generation of individual laser beams, particularly involving modeling of their physical interaction with the reflecting surfaces, can be achieved using ray tracing techniques, but is in general terribly slow and not suited for real-time operation.

The objective of this work is to simulate range measurement resulting from an ideal laser scanner, without concern for backscattering and other laser beam dynamics as it interacts with the environment. The method of depth buffer rendering is used to sample the buffer at pixels corresponding to the internal angle of the laser scanner . However, a method of frustum subdivision is presented to divide the wide field-of-view of the lidar into sections, modeling the temporal displacement of the sensor between each frustum. This offers a convenient tradeoff between modeling accuracy and performance.

6.1 Distance measurement

The distance to an object illuminated with a laser pulse can be determined by comparing the time of return to the time of emission to calculate the time of flight Δt and the resulting distance $d = \frac{1}{2}c\Delta t$, for the speed of light, c. A Lidar setup typically involves a rotating laser scanner, but we will begin with introducing how to generate a single distance measurement of a 3D scene using the depth buffer of an OpenGL camera.

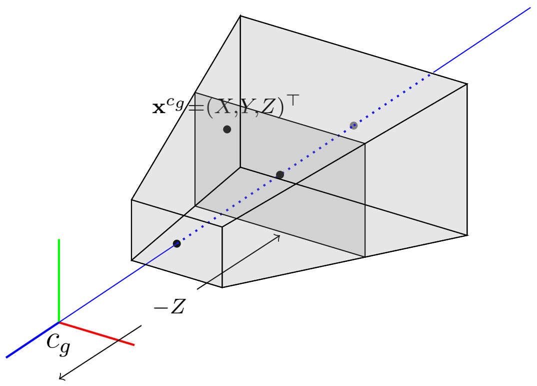


Figure 6.1: The scene depth of a point $\mathbf{x}^{c_g} = (X, Y, Z)^\top$.

Consider a point $\mathbf{x}^{c_g} = (X, Y, Z)^\top$ and the perspective transform coordinates $(x_c, y_c, z_c, w_c)^\top$, and the final normalized coordinates $(x_n, y_n, z_n)^\top = (x_c/w_c, y_c/w_c, z_c/w_c)^\top$ where $w_c = -Z$, resulting from applying the projection matrix in (5.2) to a homogeneous coordinate $(\mathbf{x}^{c_g}, 1)^\top$. Using the definition of the projection matrix it is possible to reconstruct the 3D coordinates for a sampled image point. Let $(x, y)^\top$ be a given pixel coordinate, where $x \in [0, w - 1]$, $y \in [0, h - 1]$ for the image width and height. A depth buffer contains the corresponding depth value $z \in [0, 1]$. The normalized coordinate is then $z_n = 2z - 1$. Using $z_n = z_c/w_c = -z_c/Z$ and the definition of the projection matrix (5.2) we get

$$z_n = \frac{\frac{(f_z+n)}{f_z-n} Z + \frac{2 f_z n}{f_z-n}}{Z}$$

We want to unproject the depth value to a 3D point defined in the camera frame as defined in figure 5.2. Solving for the Z coordinate in the c_g camera and employing the map $Z \mapsto -Z$ to convert from the c_g frame looking down the negative z-axis, we get a positive Z depth aligned with the c frame, according to

$$Z = \frac{2fn}{(f + n) - z_n(f - n)} \tag{6.1}$$

Z-depth is not the same as radial distance, as all points on a vertical plane a distance Z in front of the camera have the same depth Z, but differing distance depending on X, Y, as shown in figure 6.1. Using the pinhole unprojection from (5.4) we can recreate the point $\mathbf{x}^c = (X, Y, Z)^\top$ for the given Z value:

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \frac{Z}{f} \begin{pmatrix} x - c_x \\ y - c_y \end{pmatrix} \tag{6.2}$$

This is all the information necessary to create an explicit distance measurement for a single laser pulse

$$d = \sqrt{X^2 + Y^2 + Z^2}$$

6.2 2D laser scan

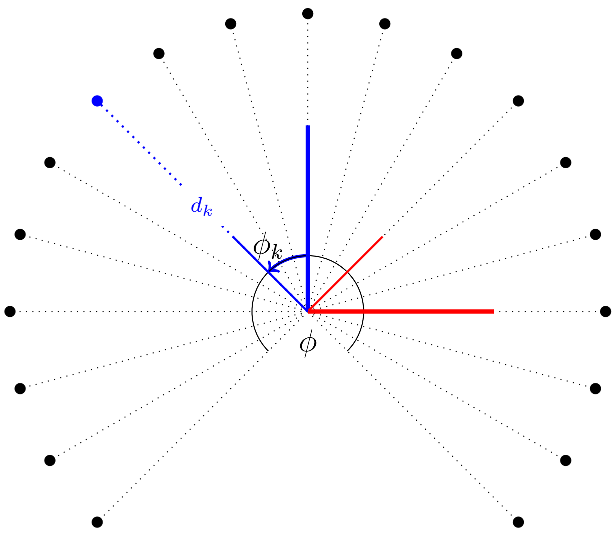


Figure 6.2: 2D laser scanner sampling points in the plane, within its horizontal field of view. Each distance measurement in frame L_i (thin) is resolved in the principal lidar frame L (thick).

A 2D lidar sampling points at equidistant angles in the plane of a rotating laser scanner is modeled as shown in figure 6.2. Assume the lidar samples K points per scan, has a horizontal field of view ϕ, a $2\pi - \phi$ dead-zone, and a given maximum range D.

The lidar distance measurement will be resolved in the lidar frame L with principal axes as shown in figure 6.2. For each sample, the scanner has rotated an angle ϕ_k and samples the point $\mathbf{x}_k^{L_k} = (0, 0, d_k)^\top$ in the internal scan frame L_k.

$$\mathbf{x}_k^L = \mathbf{R}_k(\phi_k) \begin{pmatrix} 0 \\ 0 \\ d_k \end{pmatrix}, \quad k = 0, \dots, K-1 \tag{6.3}$$

where $\mathbf{R}_k(\phi_k) = \mathbf{R}_{L_k}^L$ is described by a rotation of ϕ_k around the common y-axis. It is important to note that although the scanner is rotating quickly, each angle in a scan is not sampled simultaneously, but sequentially. It can be compared to a rolling shutter effect in many CMOS camera image sensors wherein the image sensor is scanned sequentially from top to bottom, and not simultaneously like with a global shutter typical for a CCD image sensor, like we assumed when modeling the camera. This is an important characteristic we would like to emulate when simulating a laser scan, and to make the case explicit the notation $\mathbf{x}_k^{L(t_k)}$ is used to denote a single sample resolved in the principal lidar frame, at time t_k. Be aware that $\mathbf{R}(\phi_k)$ of course only depends on the angle ϕ_k.

6.2.1 Frustum subdivision

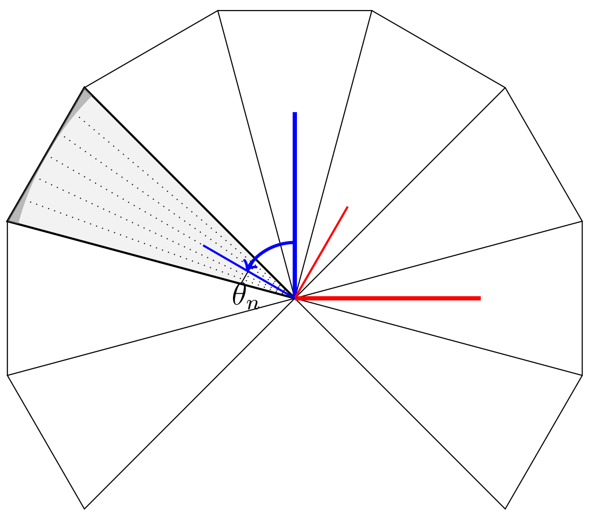


Figure 6.3: Lidar scan subdivided in frustum sections to offer temporal granularity with little overhead. Notice the darker grey edge of the frustum. This area is not strictly within the radial distance limit D from the scanner origin, and increases with the size of the frustums.

To fully emulate temporal displacement between each sample using depth-buffering, each sample would have to be constructed with its own camera, as all pixels within one image are sampled simultaneously. This is impractical for at least two reasons. First, although each rendered image scales down proportionally to the number of cameras, the overhead of rendering many separate images will cause a significant overhead in the graphics rendering pipeline, assumed not to scale well with a large number of cameras. Secondly, a lidar might typically have a scan rate of several tens of hertz, with several hundred, if not thousands samples per scan. This means a sampling rate on the order of 10 kHz. This is simply impractical to achieve in any real-time sense due to the short sampling period.

To simulate a laser scan we will therefore subdivide a scan into several equally wide sections, each corresponding to a viewing frustum to cover the entire horizontal field-of-view of the scanner. All points within a sector are sampled simultaneously, but the temporal displacement between each frustum scan is modeled. The number of frustums is an important characteristic. Too many frustums causes an overhead and performance problem as discussed. Not enough frustums and the granularity will introduce modelling errors. As is evident from the darker grey edges in figures 6.3, 6.4, the corners of the frustum is not strictly within the radial range limit D from the scanner origin. While the lidar range itself might not be strictly limited to an exact radius, in principle this modelling inaccuracy grows with the size of the frustums. Depending on the number of frustums chosen, frustum subdivision is considered a good tradeoff between modeling accuracy performance.

Let N be the number of frustums per scan. The points in each frustum are sampled from the view of a local frame L_n rotated an angle

$$\theta_n = \phi \left(\frac{k + 0.5}{N} - \frac{1}{2} \right), \quad n = 0, ..., N - 1$$

from the principal L frame. The half field of view angle $\frac{\phi}{2}$ is subtracted to offset $\theta_n = 0$ to the middle sector pointing along the principal z-axis, at least in the case of an odd number of sectors. Assuming the lidar scans with a rate of f_s scans per second, then a counting θ_{t_k} can be updated according to

$$\theta_{t_{k+1}} = \text{fmod}\left(\theta_{t_k} + 2\pi f_s \Delta t + \pi, 2\pi\right) - \pi$$

to determine which frustum, if any, is active at the current time t_k.

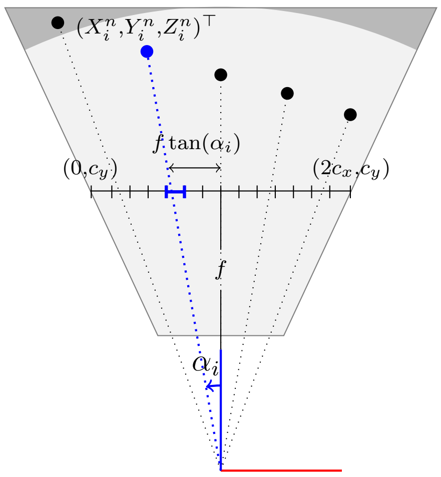


Figure 6.4: Sampling points at equidistant angles in one frustum scan, and the corresponding pixels in the depth buffer.

As shown in figure 5.2, a virtual image plane a metric f' away from the camera has a width $2d_xc_x$ and height $2d_yc_y$. For an ideal, square pixel image sensor like the OpenGL camera where $d = d_x = d_y$ it is then clear that at a distance $f = \frac{f'}{d}$ away from the camera the virtual image plane is $2c_x$ wide and $2c_y$ high, in pixel units[1]. Each frustum will have a horizontal field of view $\alpha = \phi/N$. From figure 6.4 it is then clear that for a given α and wanted horizontal pixel resolution w,

$$f = \frac{w}{2\tan(\alpha/2)} \tag{6.4}$$

where $c_x = \frac{w}{2}$. For a horizontal 2D scan, we only need to sample an image along $y = 0$, and so we can construct the frustum where $h = 1$, $c_y = \frac{h}{2}$ and the necessary frustum parameters are given from (5.5). The far plane have to be set to emulate the range limit D of the lidar. That is $f_z = D$.

[1]We will allow this combination of metric and pixel units as we are only interested in the ratio c_x/f and c_y/f, where the pixel size d is divided out.

6.2.2 Sampling

For K total samples per scan we have $M = \frac{K}{N}$ samples per frustum at equidistant angles given by

$$\alpha_i = \alpha \left(\frac{i + 0.5}{M} - \frac{1}{2} \right), \quad i = 0, ..., M - 1$$

where half frustum angle $\frac{\alpha}{2}$ is subtracted to offset $\alpha_i = 0$ to the middle of the frustum, as shown in figure 6.4. w should be chosen to have sufficiently many pixels to sample, so that each angle samples a unique pixel. In short we want $w > 2M$. For a given sampling angle α_i it is clear from figure 6.4 that the corresponding coordinate to sample is $(c_x - f \tan(\alpha_i), c_y)^\top$. Rounding off to nearest integer, between $[0, w-1] \times [0, h-1]$, the pixel coordinates to sample are

$$\begin{pmatrix} x_i \\ y \end{pmatrix} = \begin{pmatrix} [c_x - f \tan(\alpha_i)]_0^{w-1} \\ [c_y]_0^{h-1} \end{pmatrix}$$

From (6.1) the depth Z_i^n is given from sampling the depth buffer, and from (6.2) the resulting coordinates are

$$\begin{pmatrix} X_i^n \\ Y_i^n \end{pmatrix} = \frac{Z_i^n}{f} \begin{pmatrix} x_i - c_x \\ y - c_y \end{pmatrix}$$

Then, applying the rotation to transform from the frustum frame L_n to the principal frame L we get

$$\mathbf{x}_{i,n}^{L(t_n)} = \mathbf{R}_n(\theta_n) \begin{pmatrix} X_i^n \\ Y_i^n \\ Z_i^n \end{pmatrix}$$

Creating the explicit distance measurement $d_i^n = \sqrt{X_i^{n2} + Y_i^{n2} + Z_i^{n2}}$ is not necessary. However, if we did, we get the equivalent result by applying an extra rotation transform for the internal frustum sampling angle α_i:

$$\mathbf{x}_{i,n}^{L(t_n)} = \mathbf{R}_n(\theta_n) \mathbf{R}_i(\alpha_i) \begin{pmatrix} 0 \\ 0 \\ d_i^n \end{pmatrix}$$

Comparing to figure 6.2 it is clear that $\phi_k = \theta_n + \alpha_i$, and in fact $\mathbf{R}_k(\phi_k) = \mathbf{R}_n(\theta_n) \mathbf{R}_i(\alpha_i)$ corresponding to the model (6.3).

6.2.3 View matrix

The view matrix has to be set up for each individual frustum. Let the simulated state $\mathbf{r}_b^w$, $\mathbf{R}_b^w$ be given, along with a body-sensor offset $\mathbf{r}_{bL}^b$, $\mathbf{R}_L^b$. To construct the view matrix we additionally have to apply a rotation for the current frustum within the lidar frame in addition to transforming to the OpenGL camera frame c_g. Then

$$\begin{aligned}\mathbf{r}_L^w &= \mathbf{r}_b^w + \mathbf{R}_b^w \mathbf{r}_{bL}^b \\ \mathbf{R}_{c_g}^w &= \mathbf{R}_b^w \mathbf{R}_L^b \mathbf{R}_{L_n}^L \mathbf{R}_{c_g}^{L_n}\end{aligned}$$

where $\mathbf{R}_{L_n}^L = \mathbf{R}_n(\theta_n)$. Transforming a point $\mathbf{x}_p^w$ from world coordinates to camera coordinates, $\mathbf{x}^{c_g} = \mathbf{R}_{c_g}^{w\top}(\mathbf{x}_p^w - \mathbf{r}_L^w)$, and the view matrix is of course

$$\mathbf{T}_{4\times4} = \begin{pmatrix} \mathbf{R}_{c_g}^{w\top} & -\mathbf{R}_{c_g}^{w\top}\mathbf{r}_L^w \\ 0 & 1 \end{pmatrix}$$

6.3 3D laser scan

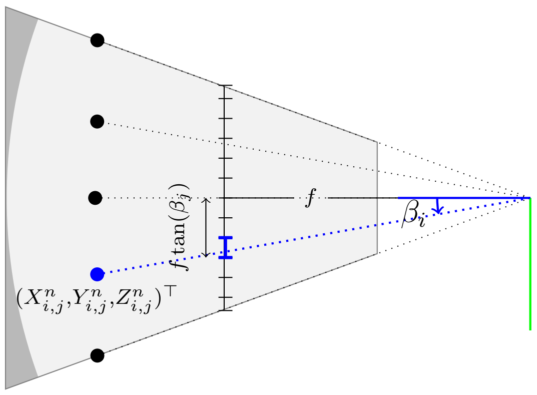


Figure 6.5: Sampling points in the vertical field of view of a laser scanner.

By extending the frustum with a less than planar vertical field of view, a 3D laser scanner is readily available. We will model the 3D lidar as a vertical array of Q scanners angled equidistantly apart over a certain vertical field of view β, each in turn working exactly like a planar 2D scanner like before.

Similarly to the horizontal field of view, we have $f = \frac{h}{\tan(\beta/2)}$. Solving for the horizontal resolution h and substituting in (6.4) we get

$$h = w\frac{\tan(\beta/2)}{\tan(\alpha/2)}$$

where $c_y = h/2$. h then needs to be scaled through w to achieve sufficient resolution $h > 2Q$, depending on β and Q. For the Q vertical arrays, the sampling angles are

$$\beta_j = \beta\left(\frac{j}{Q} - \frac{1}{2}\right), \quad j = 0, ..., Q-1$$

where again the half field of view angle $\frac{\beta}{2}$ are subtracted to offset $\beta_j = 0$ to align with the center of the frustum. Clearly, the corresponding point to sample in the image buffer is

$$\begin{pmatrix} x_i \\ y_j \end{pmatrix} = \begin{pmatrix} [c_x - f\tan(\alpha_i)]_0^{w-1} \\ [c_y + f\tan(\beta_j)]_0^{h-1} \end{pmatrix}$$

resulting in $Z_{i,j}^n$ via the depth buffer and

$$\begin{pmatrix} X_{i,j}^n \\ Y_{i,j}^n \end{pmatrix} = \frac{Z_{i,j}^n}{f}\begin{pmatrix} x_i - c_x \\ y_j - c_y \end{pmatrix}$$

Finally, the sample resolved in the principal lidar frame is given by

$$\mathbf{x}_{i,j,n}^{L(t_n)} = \mathbf{R}_n(\theta_n)\begin{pmatrix} X_{i,j}^n \\ Y_{i,j}^n \\ Z_{i,j}^n \end{pmatrix}$$

6.4 Point cloud

If the state $\mathbf{r}_L^w$, $\mathbf{R}_L^w$ is observable, which it is by definition through the simulation, the samples, $\mathbf{x}_{i,j,n}^{L(t_n)}$ in the general case of a 3D scanner, can be transformed from the moving lidar frame to the static world frame w, creating a consistent point cloud. Let the observed state $\mathbf{x}_b^w$, $\mathbf{R}_b^w$ be given. For the static body-sensor offset $\mathbf{r}_{bL}^b$, $\mathbf{R}_L^b$ we have

$$\begin{aligned} \mathbf{x}_L^w &= \mathbf{x}_b^w + \mathbf{R}_b^w \mathbf{r}_{bL}^b \\ \mathbf{R}_L^w &= \mathbf{R}_b^w \mathbf{R}_L^b \end{aligned}$$

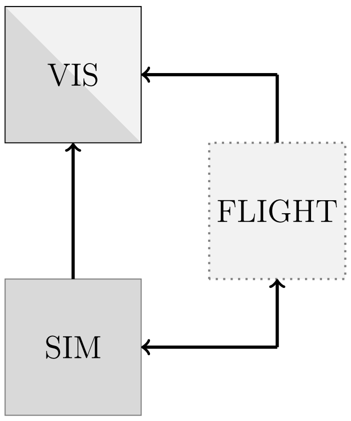
A point $\mathbf{x}_{i,j,n}^{L(t_n)}$ resolved in the world frame, as a part of a consistent world point cloud, is then given by

$$\mathbf{x}_{\mathbf{p}_{i,j,n}}^w = \mathbf{x}_L^w(t_n) + \mathbf{R}_L^w(t_n)\mathbf{x}_{i,j,n}^{L(t_n)} \tag{6.5}$$

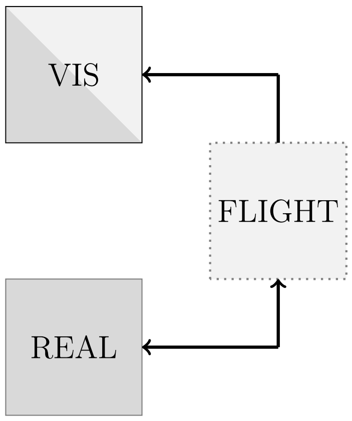
7 Implementation

The simulator is implemented as a headless program as a drop-in replacement for the inpout/output module as given by figure 1.1. It is written entirely in C++14, relying on the open source and permissively licensed libraries Eigen[9], a header-only template library for linear algebra, and OpenSceneGraph[19], a C++API for 3D graphics programming using OpenGL commonly used for visual simulation and scientific visualization and modeling.

On top of bare OpenGL, OpenSceneGraph offers a C++API and a scene graph convenient to structure the contents of a scene. Details as they relate to OpenSceneGraph specifically will not be included, as the point of the exercise is to convey the methods in terms of standard OpenGL principles. Additionally, the API specific implementation details are fairly boilerplate, and it is in the interest of this discussion to be API agnostic. Rendering of camera images and lidar depth images are done off-screen via non-default user-defined FrameBuffer Objects as explained in 7.4, 7.5. OBJ files for geometry definition, including linked materials and textures, were used to provide geometry to the virtual scene for off-screen and on-screen rendering, as well as a model of the quadrotor shown in figure 7.2 for visualization of simulation states.



(a) Flight code running as is with the simulator replacing the input/output module.



(b) Embedded flight code interacting with the real world through sensors and actuators.

Figure 7.1: Simplified overview of the interaction between the headless simulator, the *flight* code and the visualization tools. Real-time visualization is available of real and simulated sensor data, the navigational output, and the simulated states if the simulator is running.

In addition to the simulator proper, a significant amount of work have been dedicated to creating visualization tools for verifying and debugging the development of the simulator itself and the rest of the navigation, guidance and control *flight* code, through real-time visualization of internal simulator rigid body states, simulated sensor data and navigational states from flight. The visualization tools created offers a, mostly one way, graphical user interface to the overall system, including the simulator, and itself interfaces with the headless simulator and flight code through the same proprietary messaging framework. All visualization is created using OpenSceneGraph. The implementation of the tools themselves is to be considered a part of the overall work done for this thesis, but will not be described in any further detail other than stating their role in the overall system. Again, the API code is considered fairly boilerplate, and uninteresting to include in the discussion in any significant detail. The visualization will mainly be used in chapter 8 to convey the results of the simulation, which in turn will convey the results of the visualization.

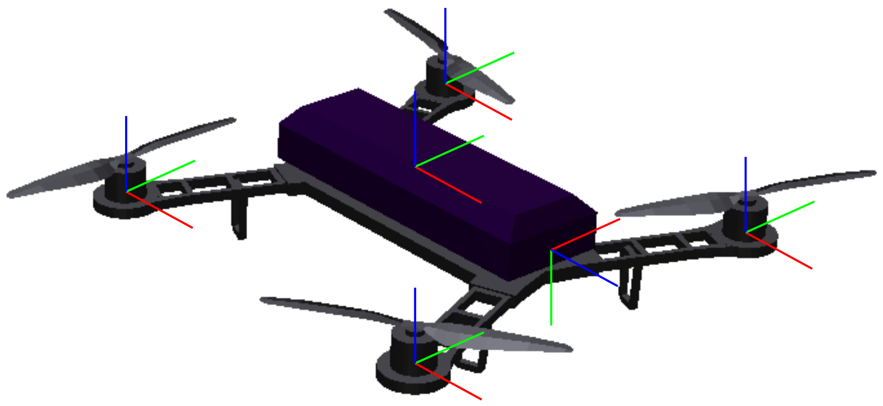


Figure 7.2: Rendering of the simulated quadrotor body. The body frame as well as poses for the actuators and camera sensor is shown.

In the simulator, each rigid body structure has a set of attached actuators and sensors as shown in figure 7.2. Additionally each body has a set of collision points to calculate rudimentary interaction with the ground, but this is not elaborated further. Also, in addition to the rotors, an actuator to calculate rudimentary wind force disturbance is included but also left out of the discussion of the overall implementation. The main loop iterates with a time step equal to the IMU sampling period given in table 7.4. For each iteration, forces and torques for each actuator is calculated according to (3.2) and (3.5) based on the latest control input, and the total actuation input is given by (3.1). Acceleration is found from (2.9), and each sensor is triggered to potentially generate data given the proper acceleration, the rigid body state in the Earth frame and the world frame, and the sensor body offset. Lastly the Earth frame acceleration is found from (2.10) and states are evolved according to the numerical integration scheme of section 2.4.

7.1 Rigid body dynamics

The WGS84 reference ellipsoid and normal gravity model was implemented with the defining parameters in table 7.1 as given by table [3, (3.1)]. The world frame is located in relation to the reference ellipsoid with a set of longitude, latitude and height coordinates $(\lambda, \phi, h)^\top$ as shown in section 2.3.3. As explained, the orientation $\mathbf{R}_w^e$ of the world frame can be set with the z-axis normal to the ellipsoidal surface, with *yaw* as a free parameter, by applying the rotation between the local gravitation $\mathbf{g}^w = (0, 0, -1)^\top$, normalized for convenience, and the global normal gravitation $\mathbf{g}_w^e$ at the point of origin. The orientation, computed as a quaternion, is readily available using Eigen:

$$\texttt{Eigen::Quaterniond::FromTwoVectors(}\mathbf{g}^w, \mathbf{g}_w^e\texttt{)}$$

Description	**Symbol**	**Value**	**Units**
semi-major axis	a	6378137.0	m
reciprocal of flattening	$1/f$	298.257223563	1
angular velocity of the Earth	ω	7292115.0e-11	rad/s
gravitational constant of Earth[1]	GM	3986004.418e8	mm^2/s^2

Table 7.1: Defining parameters of the WGS84 reference ellipsoid.

7.2 Quadrotor actuation

Without specification to a particular hardware platform, a very simple, symmetric model of a micro aerial vehicle sized quadrotor was implemented, with reasonable physical parameters given in table 7.2. The objective for the modeling here is to offer a reasonable dynamic model for a quadcopter of a certain size, with as few parameters as possible, and not necessarily the most accurate model of any given hardware platform based on system identification methods.

Description	**Symbol**	**Value**	**Units**
mass	m	1.5	kg
arm length	r	0.20	m
moment of inertia around x-axis	I_{xx}	0.05	kgm^2
moment of inertia around y-axis	I_{yy}	0.05	kgm^2
moment of inertia around z-axis	I_{zz}	0.1	kgm^2

Table 7.2: Quadrotor rigid body parameters.

Similarly, we want to arrive at a reasonable dynamic model for the rotor actuators, with hopefully a simple parametrization. We will not perform system identification in order to

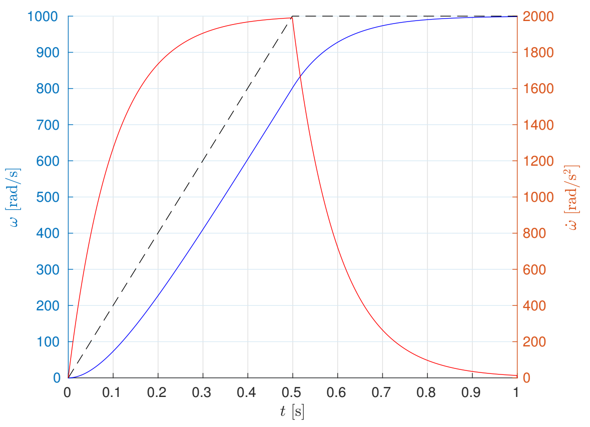
[1]Including mass of Earth's atmosphere.

best simulate a particular hardware setup. The dynamic characteristics we are interested in emulating is a first order response from control input to angular speed given by (3.4) and the resulting thrust and torque of (3.2).

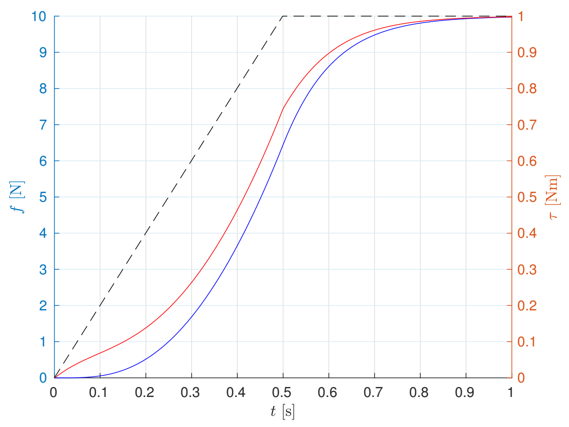
Description	Symbol	Value	Units
motor input constant	K	1000	rad/s
motor time constant	T	0.1	s
propeller force constant	K_F	1e-5	N/(rad/s)^2
propeller torque constant	K_Q	0.1	m
rotor inertia	I	5e-5	kgm^2

Table 7.3: Rotor model parameters.

We will determine reasonable values for the necessary rotor parameters given in table 7.3 from basic principles. The K_v rating of a motor is a significant characteristic. It is inversely proportional to K_e from (3.3), although with different units, i.e $K_v \sim \frac{1}{K_e}$. The K_v value is the revolutions per minute the motor needs to turn to produce 1 V of counter-electromotive force. For a quadrotor of this size with 8-9 inch propellers, a typical motor used might have a $K_v = 1000$ rating, achieving approximately 10 000 RPM at maximum throttle, depending on the battery voltage and the load. When 10 000 RPM $\approx$ 1000 rad/s, from (3.4) at stationary behaviour we see that $\omega_{\max} = Kr_{\max}$, $r_{\max} = 1$, so a reasonable value is $K = 1000$rad/s. If we want this maximum throttle to correspond to for example 10 N, this leads to $K_F = 1 \times 10^{-5}$ N/(rad/s)^2. K_Q should equal the moment arm of the torque, the propeller diameter, up to a scale factor depending on C_Q/C_F. Typically, for a 0.2 m propeller diameter, $K_Q = 0.1$ m. The approximate moment of inertia I for a small propeller 0.2 m diameter is typically[12] 5×10^{-5} kgm^2. The first order response time was simple guesstimated to have a time constant on the order of 0.1 s.



(a) Angular speed (blue) and angular speed rate of change (red).



(b) Force (blue) and torque (red) as a result of drag and angular speed rate of change.

Figure 7.3: Transient rotor output in response to an increasing input (dotted black). MATLAB implementation with parameters from 7.3.

7.3 Inertial measurements

The particular accelerometer and gyroscope that was implemented in this was modeled after the ADIS16488[7] inertial measurement unit. It has an internal sampling rate of 9.84 kHz, and an averaging/decimation filter which reduces the update rate to 2.46 kHz. The user can choose an additional decimation rate to discretely sample this signal at potentially decreased rates. We choose a sampling rate of 1.23 kHz.

The sampling causes discrete measurement white noise, proportional to the square of the sampling period[26]. The data-sheet provides output noise standard deviations of 1.5mg and 0.16deg/s for the accelerometer and gyroscope respectively, where g is the standard gravity. While it is uncertain what the corresponding sampling rates are, we will use the given output noise as standard deviations in our discrete white noise models, with the disclaimer that the actual discrete sampling error deviation might be off by a factor of $\sqrt{2}$. The final parameters are given in table 7.4. The biases are modeled as first order Gauss-Markov processes with large time constants, with parameters guesstimated in the same way they are used as noise parameters in the navigation filter.

Description	**Symbol**	**Value**	**Units**
acc discrete noise std	σ_a	0.0147	$\mathrm{m/s^2}$
gyro discrete noise std	σ_ω	0.0028	rad/s
acc bias driving noise std	σ_{b_a}	1e-5	$\frac{\mathrm{m/s^2}}{\sqrt{\mathrm{Hz}}}$
gyro bias driving noise std	σ_{ω_a}	1e-5	$\frac{\mathrm{rad/s}}{\sqrt{\mathrm{Hz}}}$
acc bias time constant	T_{b_a}	1e3	s
gyro bias time constant	T_{b_ω}	1e3	s
sampling frequency	$1/\Delta t$	1230.0	Hz

Table 7.4: Accelerometer and gyroscope parameters.

To implement the generation of the measurements, the normal distribution feature of the C++Standard Library added in C++11 is used. To model $n \sim \mathcal{N}(0, \sigma^2)$ simply declare `std::normal_distribution<> n {0,σ}`. Then draws are done according to [6] using a Mersenne Twister pseudorandom number generator. The accelerometer and gyroscope biases are initialized with a draw from a standard distribution with a standard deviation of 1.0 deg/s and 0.05 m/s^2, respectively. Finally, as the exact location of the accelerometer sensor in the IMU is unknown, the internal offsets $\mathbf{r}_x^s$, $\mathbf{r}_y^s$, $\mathbf{r}_z^s$ were simply set to zero.

7.4 Camera images

To render an image off-screen, OpenSceneGraph was used to create a windowless OpenGL graphics context. The image is stored in 2D array image buffer with the greyscale pixel format `GL_LUMINANCE` of type `GL_UNSIGNED_BYTE`. The image buffer is attached to a `GL_COLOR_ATTACHMENT` of the Framebuffer Object[29] rendered to by the camera viewing the scene, instead of the Default Framebuffer drawing to screen.

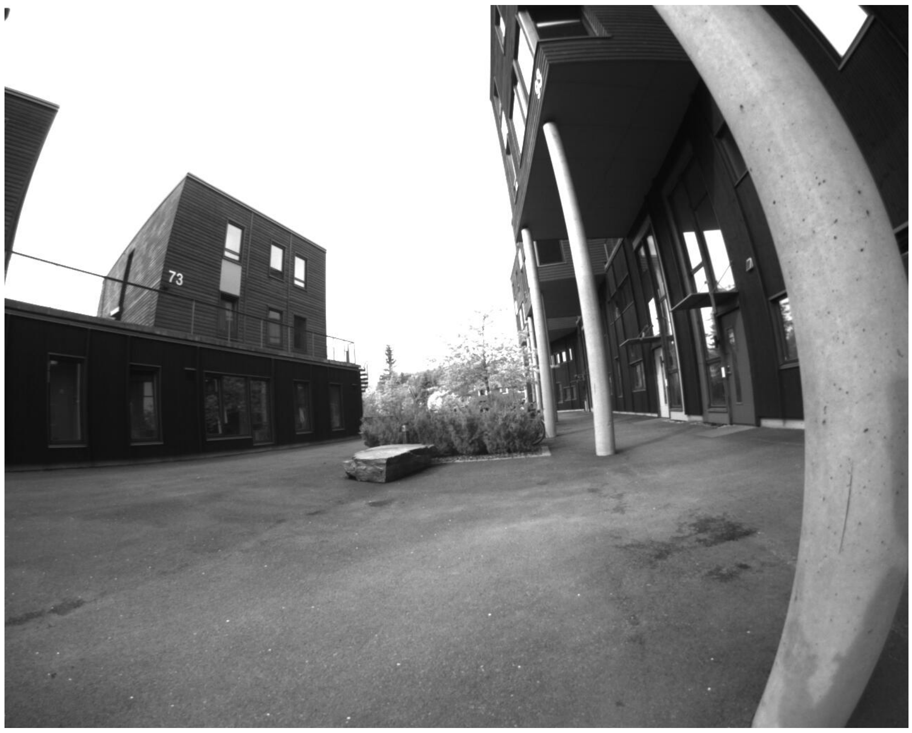


Figure 7.4: Image taken with the IDS camera. Heavy distortion resulting from the wide-angle lense.

The particular camera that was simulated is the IDS UI-3241LE, with a wide-angle fisheye-like lense producing distorted images as seen in figure 7.4. The camera produces greyscale images at $w \times h = 1280 \times 1024$ resolution, at up to 60 frames per second. Using the fisheye camera model[18], the camera was calibrated with OpenCV with distortion parameters given in table 7.6 and intrinsic parameters given by the *real* column in table 7.5.

Description	**Symbol**	**Ideal**	**Real**
focal length, horizontal units	f_x	225	5.5910e2
focal length, vertical units	f_y	225	5.5954e2
principal point offset, x-axis	c_x	1280	6.3992e2
principal point offset, y-axis	c_y	1024	5.2171e2

Table 7.5: Camera projection parameters in pixel units.

The real and ideal projections correspond to $\mathbf{h}_r$ and $\mathbf{h}_i$ from (5.10), respectively. The ideal parameters are used to specify the perspective frustum of the OpenGL camera. In principle, $\mathbf{h}_i$ is arbitrary, as the generated image is projected according to $\mathbf{h}_r$ through the (un)distortion map. However, the application of the fisheye distortion maps pixels towards the center of the image. When the undistorted and distorted images are the same size this creates a significant black border around the edges as shown in figures 7.5, 7.6, where the pixel value is undefined outside the range of the sourced undistorted image. When generating an ideal OpenGL image to source from, the field of view and resolution of the source image can be artificially enhanced. Scaling down the focal length achieves

Symbol	**Value**
k_1	-1.5363e-2
k_2	1.2678e-2
k_3	-1.2716e-2
k_4	-1.5363e-2
a	0.0

Table 7.6: Camera distortion parameters.

a greater field of view, allowing more of the scene to be viewable and reducing the black border. However, without increasing the resolution correspondingly there are significant aliasing issues. The ideal parameters of 7.5 reflects this, with a $\frac{1}{2}$x reduction of the focal length in addition to a 4 x increase in resolution as a result of doubling c_x, c_y.

Figure 7.5 shows the application of direct undistortion and subsequent distortion to the original IDS image from 7.4. They both show noticeable black line artifacts disturbing the image resulting from the inability of the direct mapping from source pixels to smoothly cover the destination image. Therefore, indirect methods as shown in figure 7.6 were implemented in practice to simulate camera images. The necessary undistortion map is precomputed at startup, employing Gauss-Newton iteration to find the best fit $(u, v)^\top$ such that $(u, v)^\top = \mathbf{f}(u', v')$, where $(u', v')^\top = \mathbf{h}_r^{-1}(x', y')$ for every destination pixel $(x', y')^\top$ as described in chapter 5. For the fisheye-distortion function, the Jacobian is given without further justification

$$\mathbf{J_f} = \begin{pmatrix} \partial f_1/\partial u & \partial f_1/\partial v \\ \partial f_2/\partial u & \partial f_2/\partial v \end{pmatrix} = \begin{pmatrix} u^2 B + A & uvB \\ uvB & v^2 B + A \end{pmatrix}$$

where

$$A = \frac{\arctan(r)}{r}, \quad B = \frac{1}{r^2(r^2+1)} - \frac{A}{r^2}$$

The iteration can be started with $(u', v')^\top$ as the best guess, or a numerical approximation of $\mathbf{g}_0^{-1}(u', v')$ for faster converge times. The camera frame rate was implemented at 20 fps, meaning one in about 60 simulator steps will result in a camera trigger. When a frame needs to be rendered, it takes significantly longer than the time step Δt for the calling function to return. Therefore the drawing of a frame is implemented as a background thread to support non-blocking operation. The camera allows synchronization with the IMU by calling a callback at every trigger stamping the corresponding frame id into the stream of inertial measurements.

7.5 Laser range measurements

To construct distance measurements, depth values from the scene are rendered to a depth buffer. Similarly to creating the camera images, a windowless OpenGL graphics context was created. The depth values are stored in a $w \times h$ array with the depth pixel format `GL_DEPTH_COMPONENT` of type `GL_FLOAT`. The depth buffer is attached to the `GL_DEPTH_ATTACHMENT` of the Framebuffer Object[29] rendered to by the camera viewing the scene.

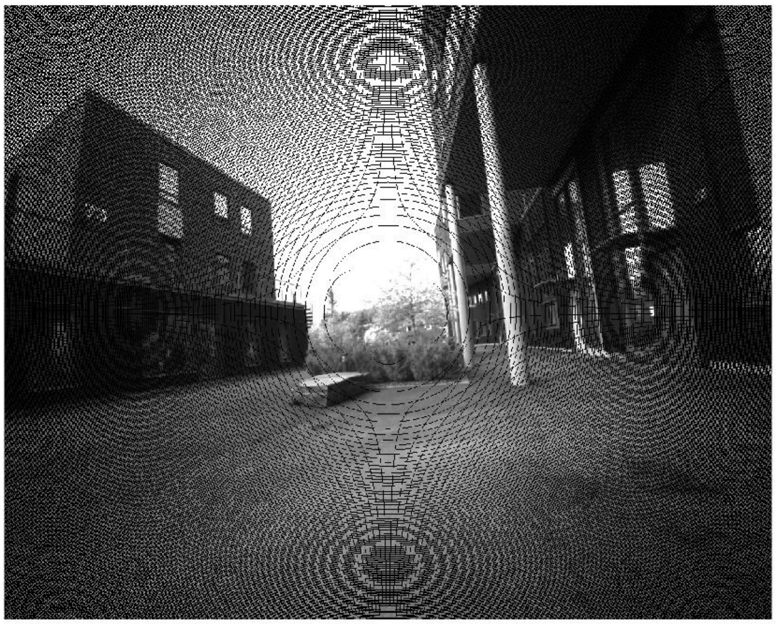
The particular lidar that was simulated is the Hokuyo UST-20LX. It is a planar 2D laser scanner, with parameters as given by table 7.7. The method described in chapter 6 is general in that it allows the construction of laser scanners with arbitrary horizontal and vertical field of view, but the scope of the final implementation is limited to the 2D planar case modeled as given in table 7.8. Similarly to the camera images, the drawing of a lidar depth image is scheduled to a background thread to support non-blocking operation of the main loop.

Description	**Symbol**	**Value**	**Units**
horizontal field of view	ϕ	270	deg
samples per scan	K	1080	1
maximum detection range	D	20	m
scanrate	f_s	40	Hz

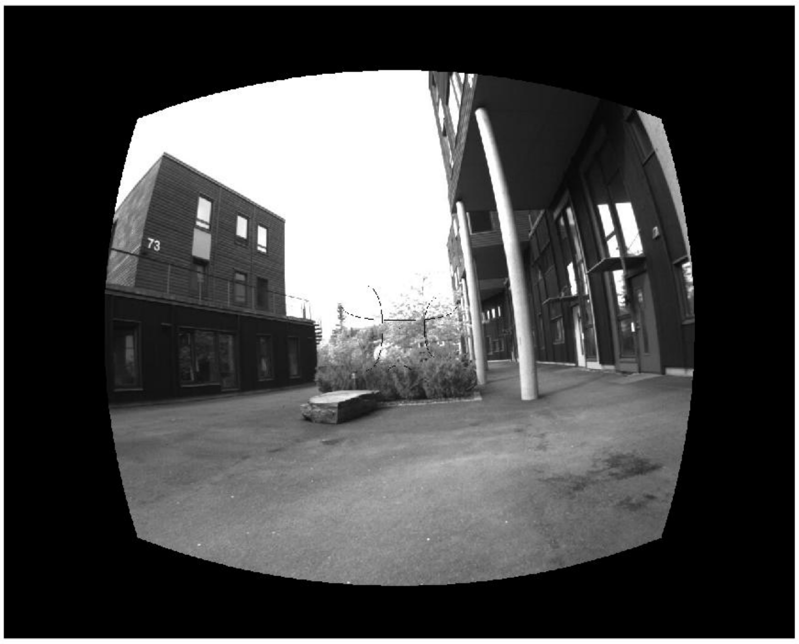
Table 7.7: Lidar parameters.

Description	**Symbol**	**Value**	**Units**
number of frustums	N	9	1
field of view per frustum	α	30	deg
samples per frustum	M	120	1
horizontal resolution	w	256	px
vertical resolution	h	1	px

Table 7.8: Lidar model parameters.

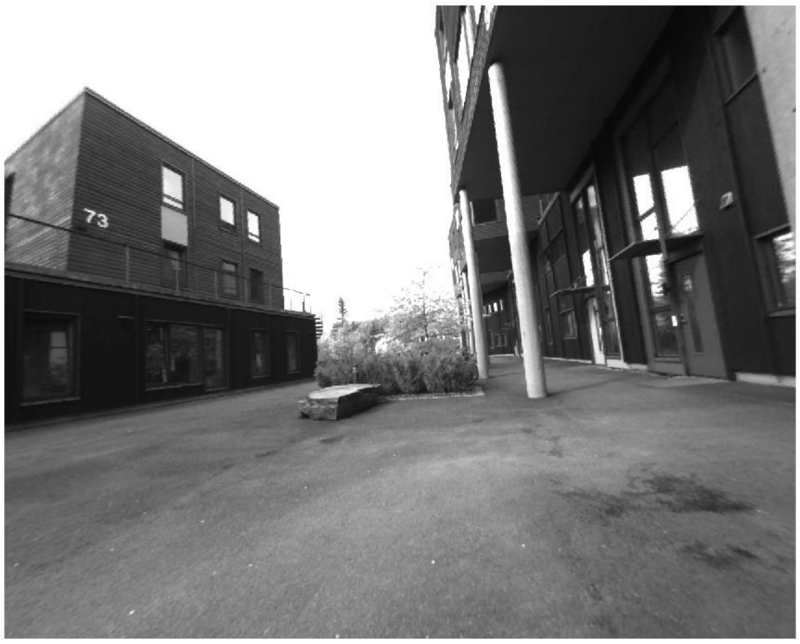


(a) Undistorted image with very noticeable black line artifacts.

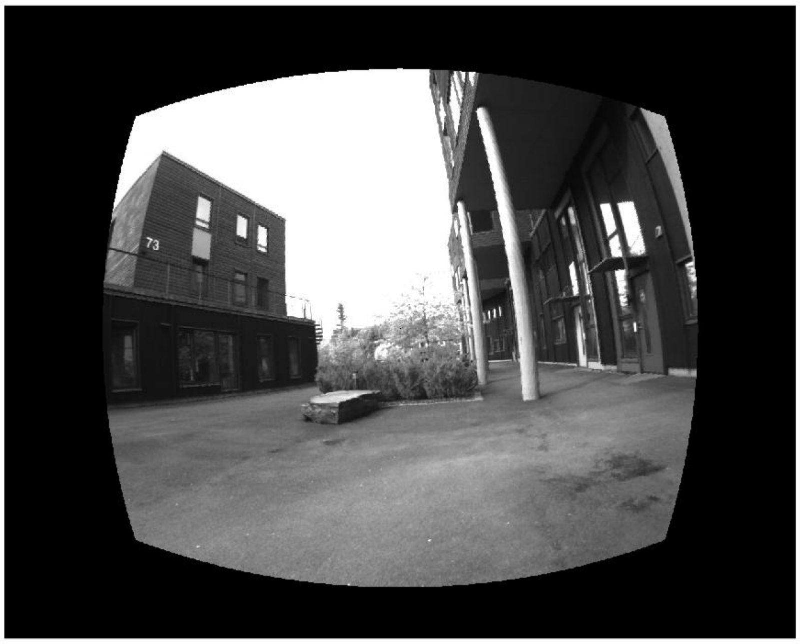


(b) Distorted image with a faint black line artifact in the middle of the image.

Figure 7.5: The original image undistorted (top) and subsequently distorted (bottom) using direct distortion methods. The direct method causes black lines to appear in the images.



(a) Undistorted image.



(b) Distorted image.

Figure 7.6: The original image undistorted (top) and subsequently distorted (bottom) using indirect distortion methods.

8 Results

We present the results from the simulator as it currently stands. The focus will be to showcase how the simulation of vehicle dynamics and sensor data successfully have been used in the development of navigation, guidance and control algorithms. This is purposefully a qualitative overview of the results rather than a quantitative analysis of all data generated as this is considered the most effective way to document and convey the validity of the results.

8.1 Vehicle dynamics

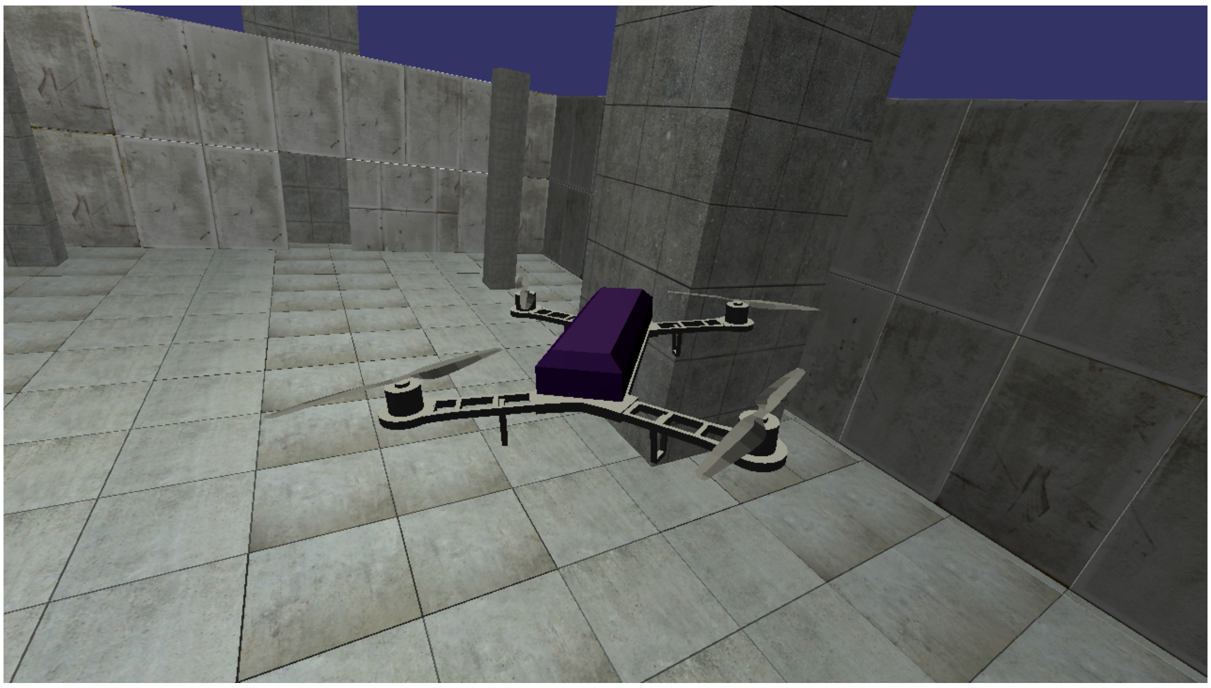
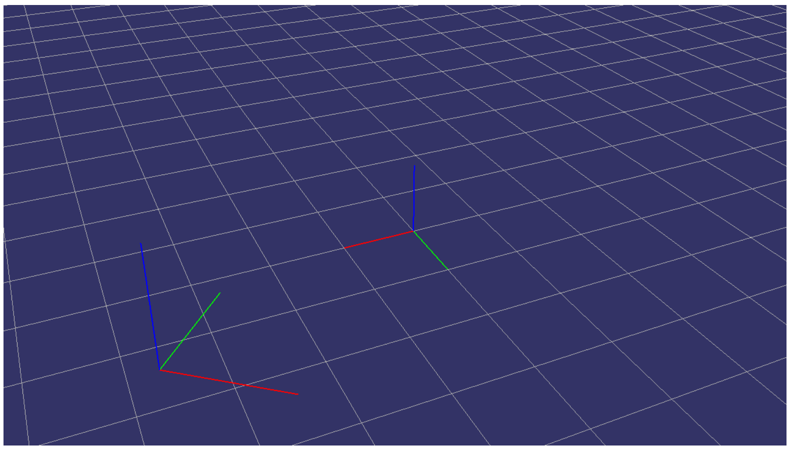


Figure 8.1: Real-time rendering of the quadrotor flying in a virtual scene.

The simulated sensor data can be used as input to the navigation filter to make state estimates available for motion control feedback. However, clearly this information is already available from the underlying rigid body dynamic simulation. This means, through simulation of vehicle dynamics alone, herein including the quadrotor actuation model and rigid body dynamics, the simulator can close the loop without generating sensor data and replace the navigation module by simply spoofing its output directly.


(a) The current pose estimate, and initial frame.

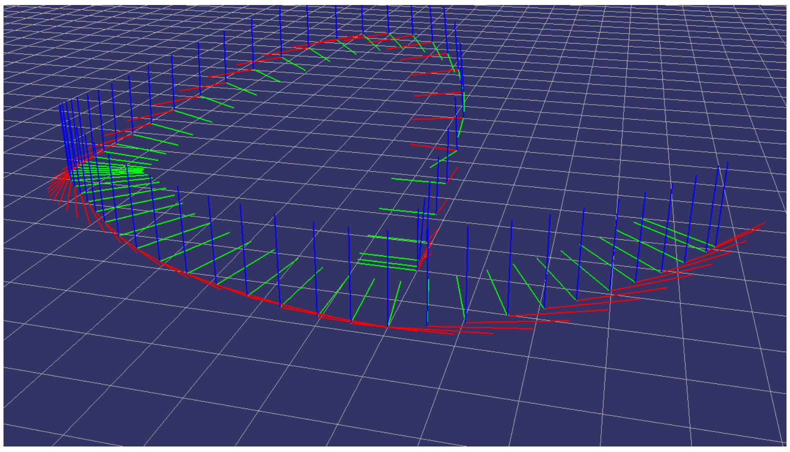

(b) A track of current and all past poses.

Figure 8.2: Navigational estimate of the pose of the body frame relative to the initial frame when the module was started. If the rigid body is initialized aligning with the world frame, then the pose outputs reveal the state in the world frame[1].

In section 1.2 it was explained that the underlying IO-board being replaced by simulation contains low-latency closed-loop control of the actuators to stabilize the angular rate of the quadrotor. This code was reimplemented in the simulator, but left out of the implementation discussion as it pertains to application specific throttle mixing. The satisfactory tuning of the real angular rate controller will filter out a lot of imperfections in the platform specific dynamics and offer a more ideal rigid body able to track a given set of angular rates at a bandwidth an order of magnitude higher than that of the higher-level velocity and attitude controls. While the actuation model was not made to emulate the specific dynamics of any particular quadrotor, this means that with a rate controller tuned for the specific underlying platform, whether real or simulated, the resulting dynamic response of the rigid body is very comparable in the real and the simulated case.

In fact, this was verified during the first flight and total system integration test of the overall system. Higher-level control gains were tuned against the simulated body and used *as is* as a very successful starting point during the first flight, requiring minimal adjustment to optimize performance against the real platform. It is important to disclaim that any work on the *real* system is not to be considered a part of this thesis work. But the two are of course related, and it is mentioned as qualitative evidence of the potential effectiveness of simulation. While neither the purpose of the flight was to log flight dynamic data, nor the purpose of the simulator to emulate a particular flight dynamic, any potential quantitative comparison between the two are considered outside the scope of this discussion.

8.2 Camera images

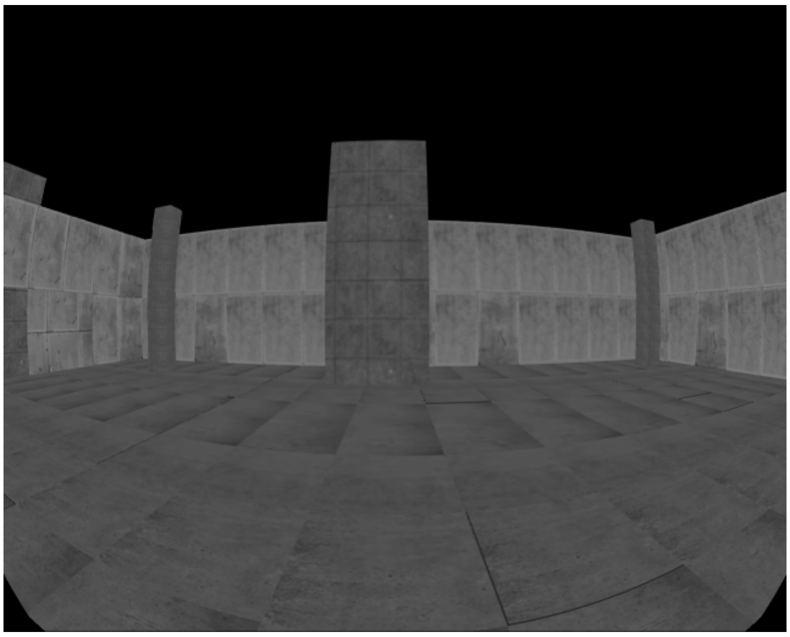
The camera images are not directly input to the navigational filter, but fed through a *tracker* running hardware accelerated computer vision algorithms to find features and track them between frames. It is very important to disclaim that any work with the tracking is not done by the author and should not be considered in any way a part of the work involved in this thesis. The visualization of the tracking is used here only to document the validity of the generated camera images and explain their use in the ongoing development of the overall system.

The focus of this implementation has not been performance. The simulator does not have any strict demands for real-time performance as it does not operate in absolute time but in nominal discrete time intervals, and there is no interaction with any physical system with actual real-time demands. However, for the human interfacing with the simulator, particularly through live visualization, it is very *convenient* if the execution is able to perform approximately real-time in order to deliver live results as they are expected. We will discuss performance in short, here.

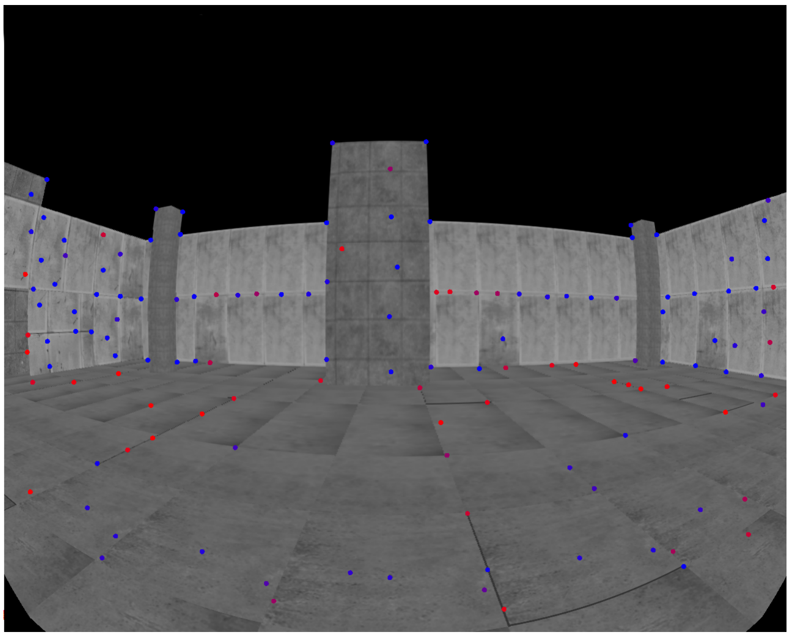
It is not straight-forward to measure rendering execution times in the OpenGL pipeline as things are not necessarily drawn when the call to draw is scheduled or finished when the call returns. The graphics card driver maintains a FIFO buffer at the front end of the graphics card where from the application point of view a draw is finished when the driver returns from the buffer. The total rendering execution time then depends on how fast the CPU can feed the pipe with data, the bandwidth of the data bus between the CPU and the graphics card buffer, and the actual vertex and fragment shader processing.

The rendering time depends on the complexity of the scene, for example rendered with or without lighting calculations. Although the executions times were not directly observable, it was experienced that the main crux in rendering times were the size of the undistorted source image. To generate the 1280×1024 pixel image as shown in 8.3, a 2560×2048 images of four times the pixel count was rendered, as explained in the implementation details in order to reduce the black border around the edges with minimal aliasing. This not only effects the GPU execution time, but the CPU time for each frame, as the

[1]When the state '*estimates*' are given by the simulated navigation module the rigid body states are the world frame is used directly, and so the world frame and the initial local navigation frame align by definition



(a) Camera image from a front-facing camera attached to the body.



(b) The same camera image overlaid tracking information.

Figure 8.3: Simulated camera images with fish-eye distortion. Features tracked with computer vision algorithms from frame to frame is shown overlaid in figure 8.3b. New features are colored in red and progressively turn blue.

number of values that have to be interpolated increase as well. On average the graphics rendering was timed to take approximately 30 ms and the CPU distortion-mapping and interpolation 20 ms on the relevant desktop workstation. This means 20 images per second is at the very limit of what the simulator can provide in soft real-time. Reducing the size of the source image significantly reduces the execution times, but produces a black border in the distorted image that is not optimal for the tracker performance. It is seen that the execution times can benefit from a hardware accelerated GPU implementation of the pixel-by-pixel distortion mapping, as the CPU execution times contribute to a significant part of the total execution time for each frame.

8.2.1 Validation by navigation

We will show the validity of the generated camera images and inertial measurements by documenting their use as input to a visual-inertial sensor fusion filter. Again it must be disclaimed that the work on the filter is in no way the work of the author. Figure 8.4 shows the estimated trajectory from a simulated run, post-processed in a MATLAB implementation of the filter. The output is very comparable to that of real data, with the same *consistency* and *accuracy* of results. In addition to offline MATLAB processing, the C++implementation of the filter was successfully run live using generated sensor data from the simulator.

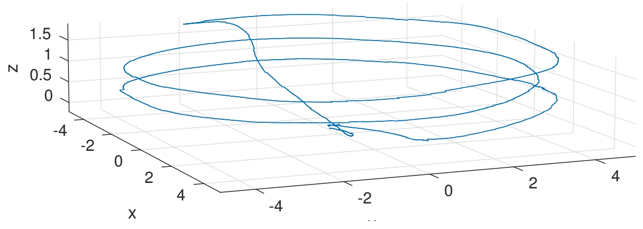
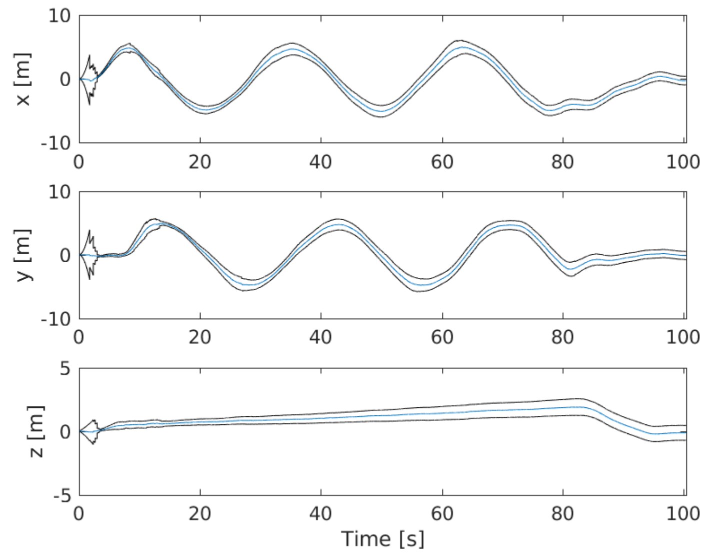
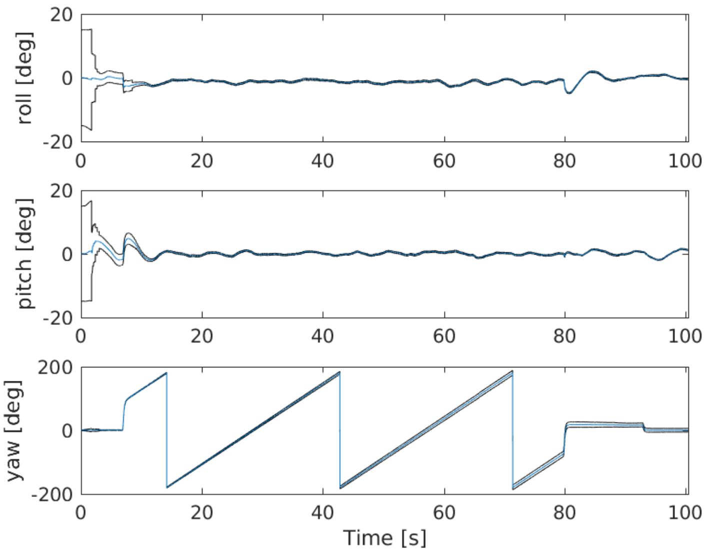


Figure 8.4: Trajectory of estimated positions from a 100 s long simulation run. The results were post-processed in MATLAB from logged inertial data and tracked features.

The filter is model-free, that is it does not assume anything about the dynamics underlying the sensor data, except of course the fundamental laws of kinematics. Model-free algorithms, then, can be tested in isolation from any vehicle dynamics. An implementation can be tested using real logged data. While verifying filter performance using real data is crucial, apparent drawbacks during development include that the hardware integration needs to be available, the practical overhead of logging several runs to generate statistically verifiable results, and the potential inconsistencies in the data which might interfere in the analysis of the filter. In addition data is only fed into the filter without the ability to use the estimation output as feedback to control the motion generating the sensor data, and closing the loop. This is crucial for the full integration test of the entire system, where navigation, guidance and control are interdependent. Without a field test this is only possible with simulated sensor data in combination with simulated vehicle dynamics closing the loop between control input and navigation output.



(a) Estimated position.



(b) Estimated orientation, converted to Euler angles. Notice the yaw wraps to $[-180°, 180°]$.

Figure 8.5: Estimated position and orientation from the same run shown in 8.4. The best estimate is shown in blue while a 3 σ uncertainty envelope is shown in black.

8.3 Laser range measurements

The 2D planar lidar was implemented. A visualization of the range measurements from a scan is shown in figure 8.6. What is shown is not directly the output from the lidar data, which is resolved in the lidar frame at the time of each frustum rendering. Rather the measured points were converted to the world frame, which is consistent between scans, at the time of generation using the body frame pose directly available from the rigid body simulation to output a point cloud valid in the world frame, according to (6.5). This is done to make the visualization pixel perfect, as evident in the figure where the points visualized as 4 pixel wide dots are visible from both sides of the walls modeled as a 2D quad with zero depth. This is turn confirms the correctness of the lidar data, for each frustum and for each scan.

For the lidar parameters given in table 7.8, for a 40 Hz scan rate, 9 live and 3 dead zone frustums per scan each frustums is allotted an average of approximately 2000 µs. A timing of the drawing calls as seen from the CPU revealed an average execution time of well under 1000 µs, depending on the complexity of the scene and the other parallel rendering jobs in the pipeline. This means a frustum subdivision on the order of 10 frustums are acceptable, verifying the method. However, it was seen that the execution times scaled poorly with a significant amount of frustums per scan, presumably as the frame rendering is not the bottleneck but rather the added overhead of piping many small jobs to the graphics card buffer. We then see that the method of conservative frustum subdivision offers a a nice trade-off between modelling accuracy and performance.

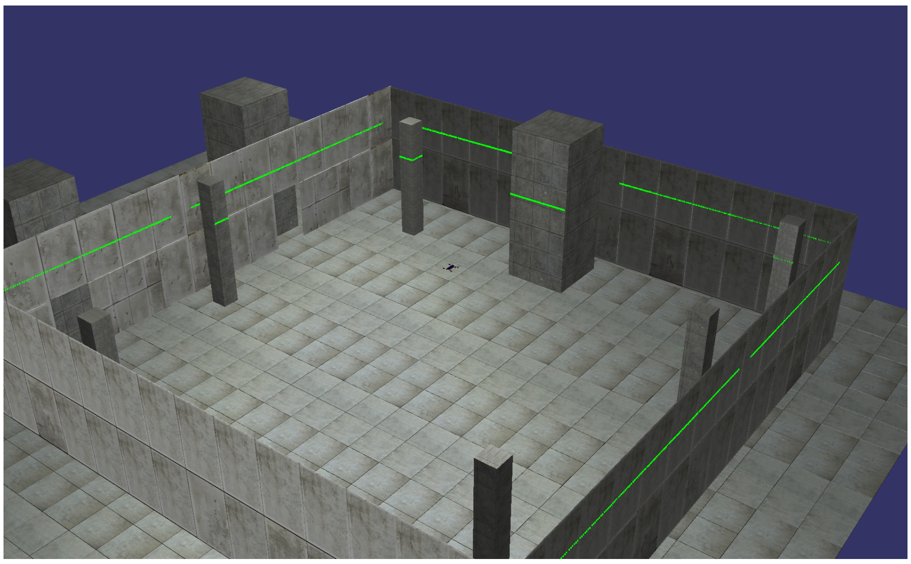
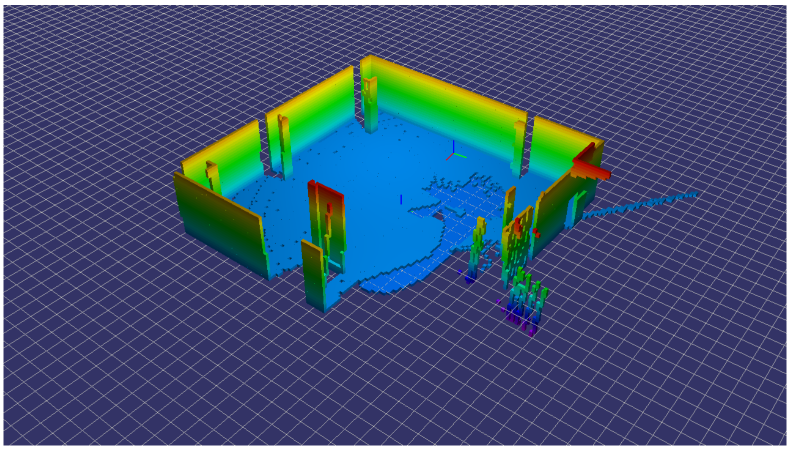


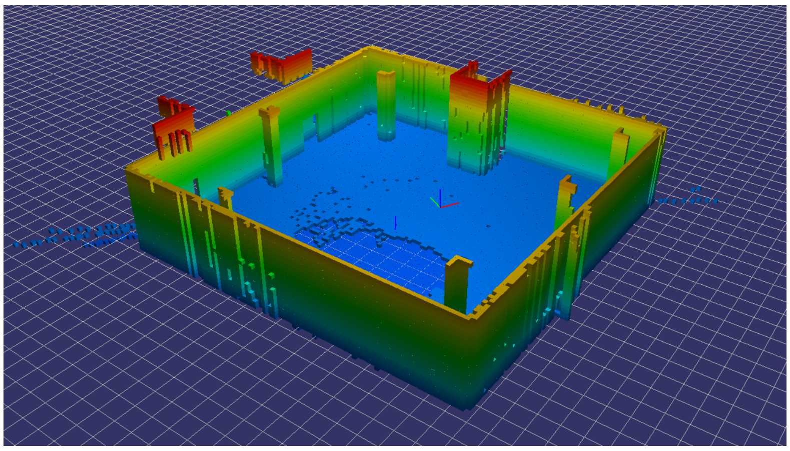
Figure 8.6: Visualization of lidar point cloud, in green, resulting from a single scan. The point cloud was resolved in the world frame, using the body frame pose directly available from the rigid body simulation. The result is a pixel-perfect point cloud.

8.3.1 Occupancy map

Compared to the point cloud data, the actual output data from a simulated lidar scan was used to update a occupancy map using the open source OctoMap library[17], for demonstrational effect to validate the generated lidar data. The OctoMap library implements an octree datastructure and algorithms to update an occupancy grid in a probabilistic fashion to establish a map denoting free, occupied and unknown spaces. We will not document the implementation in any detail, only mention that occupancy mapping is a very real use case for actual lidar data and an effective tool for any autonomous robot. Generating and updating occupancy maps in real-time is essential in order to test guidance algorithms based on dynamic route planning in any given scene, and the simulation of lidar data and vehicle dynamics offers this possibility from the desktop.



(a) Partial map before the area has been sufficiently scanned.



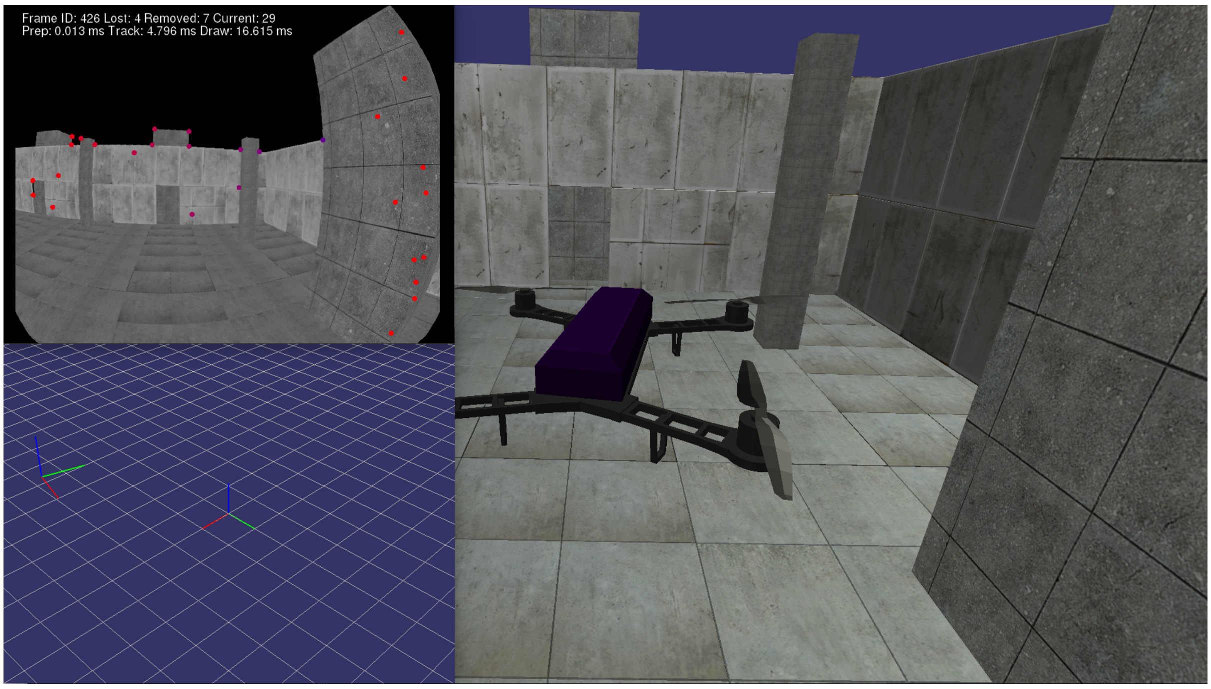
(b) A more complete map after all areas have been scanned.

Figure 8.7: Real-time rendering of the occupancy map generated with lidar measurements[2]. The lidar frame measurements are transformed to a consistent frame based on the current estimate of the body frame pose.

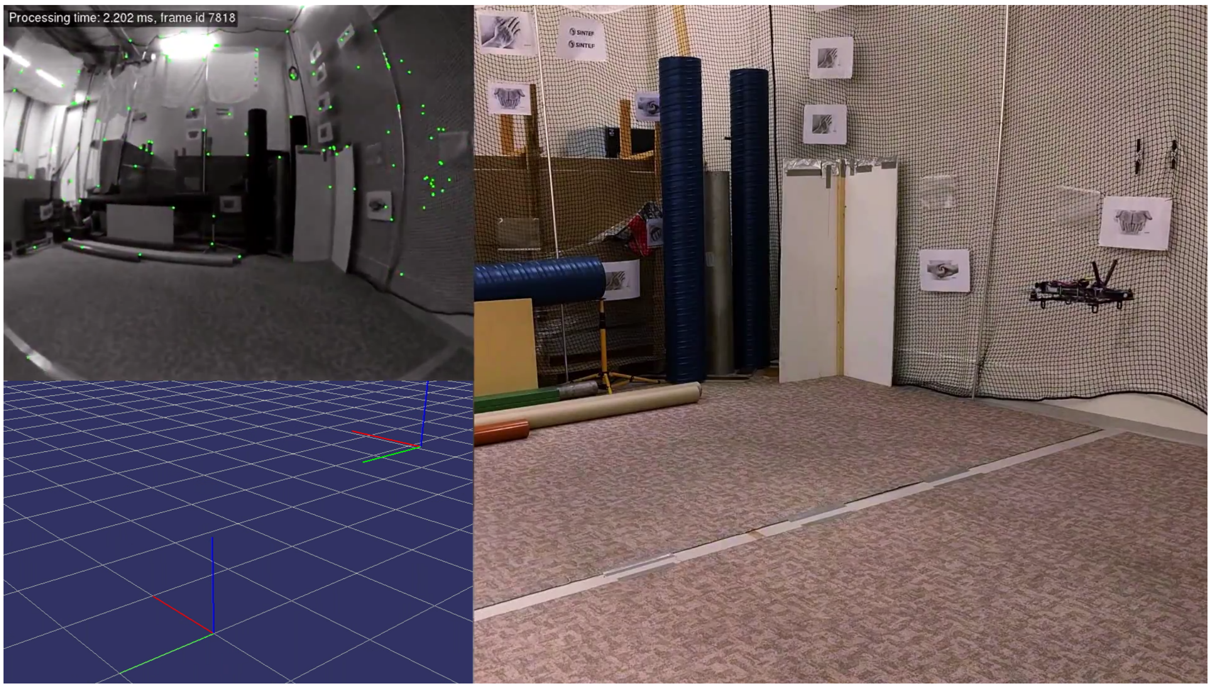
[2]The scans used to build the maps shown were generated with a vertically oriented lidar, in contrast to what is visualized in figure 8.6.

8.4 Final remarks

As it stands, the simulator has been used with good effect during the development of embedded flight control and navigation algorithms for an autonomous quadrotor. Through extensive use of the simulator, control parameters were tuned, navigation algorithms were verified and guidance programs were tested. In combination with independent testing and verification of the embedded IO module and corresponding IO board, and all necessary hardware integration, the first ever full flight test of the entire system was performed successfully, demonstrating autonomous navigation indoors, without motion capture or any other external signals. It is important to reiterate that the overall work done on the system should not be considered work done by the author as a part of this thesis. However, it bares mentioning as it demonstrates the effectiveness of the results. In figure 8.8 we show a comparison between a simulated flight and the aforementioned first ever real flight to sell the idea of the one-to-one correspondence between actual flight and simulated flight.


Frame ID: 426 Lost: 4 Removed: 7 Current: 29
Prep: 0.013 ms Track: 4.796 ms Draw: 16.615 ms

(a) Simulated flight.



(b) Real flight.

Figure 8.8: Side-by-side comparison of a simulated run versus a real run. Each overlaid with the camera images including features (top-left) and navigational pose estimate (bottom-left). During the real run only the pose estimates were available live for real-time visualization due to wifi band-width constraints. The camera images were logged live and played-back offline.

9 Conclusion

For a navigation, guidance and control system running on Linux, a desktop simulator like the one presented in this thesis has proven to be an effective development tool wherein only a minimal part of the system involved with interfacing with the real world through sensors and actuators is replaced with a simulation of the hardware and vehicle dynamics. The rest of the system can run as it would on the embedded platform and benefit from the simulator providing valuable live feedback for testing and further development. Using network-based communication the simulator can also interface with the system running on the embedded platform to provide final validation of the compiled binary and dynamically linked libraries completely *as is* before field testing.

For an autonomous mobile robot dependent on acquiring information about its environment, through camera imagery and laser range measurements, it was shown that graphics programming through the use of OpenGL is a readily available strategy. An OpenGL camera is easily constructed from a set of intrinsic camera parameters, and the effects of optical distortion of a wide-angle lense was shown to be emulated with an *indirect* application of the inverse distortion function. Utilizing the definition of the OpenGL projection matrix, depth buffer values can be unprojected into three-dimensional coordinates to provide range measurements. An array of OpenGL cameras was arranged to simulate the range measurement data from a typical scanning lidar, sampling the depth buffers accordingly to emulate scanning samples at equidistant angles.

While the sensor data can be used as input to a navigation filter to estimate state available as feedback input for the motion control, it is clear that this information is already perfectly available from the rigid body dynamics. This means, through dynamic simulation alone the simulator can close the loop without involving navigation or generating any sensor data. This alone was shown to be a valuable tool to be used during the development and testing of guidance and control algorithms, where the visual tools were used extensively.

Model-free navigation algorithms can be tested in isolation from any vehicle dynamics, with either logged or real-time data. While it is crucial to test with real data in order to validate the results, it is also limited as the hardware sensor setup needs to be available, there might be inconsistencies in the data which interferes with the validation of the filter, and there is no integration of the estimation output as feedback to the rest of the system. Without a field test, this is only achievable with simulated sensor data and simulated vehicle dynamics closing the loop between control input and navigation output. While we see that it is very constructive to test navigation algorithms in a variety of ways, including with real and simulated data, we conclude that the proposed simulator is an indispensable tool in the development of vision-based autonomous robots.

9.1 Further work

- Currently, the simulator is limited to provide inertial measurements, camera images and lidar measurements, to facilitate local navigation based on visual-inertial sensor fusion not reliant on external positioning signals. However, when global navigation satellite signals are available they provide excellent information of the global position. A worthwhile extension would be to simulate a receiver for a global positioning system. The fundamentals of such an implementation are already in place through the WSG84 reference ellipsoid.
- The simulator is currently restricted to quadrotor actuation models only. This is reasonable as the controls being tested are limited to quadrotors as well for the time being, but seeing as the system in large part is model-free, including the navigation and guidance as well as rigid body dynamics and the sensor data being simulated, it seems reasonable to extend the simulator with additional mobile robots actuation models. For example that of a fixed-wing aerial vehicle.
- While the undistortion map is computed offline, the process of mapping and interpolating every pixel in every image is time consuming, and at 20 images per second it is pushing the performance limits of the development desktop computer. Performance has not been the focus of this work, but it is clear that the pixel by pixel image distortion operation could benefit from hardware acceleration in form of a GPU implementation.

References

[1] Song Ho Ahn. *OpenGL Projection Matrix*. [Online; accessed 19-May-2017]. 2017. URL: http://www.songho.ca/opengl/gl_projectionmatrix.html.

[2] JG Balchen, T Andresen, and BA Foss. "Reguleringsteknikk Department of Engineering Cybernetics". In: *Norwegian University of Science and Technology, Trondheim, Norway* (2003).

[3] MJ Boyle. *Department of Defense World Geodetic System 1984-It's definition and relationship with local geodetic systems*. Tech. rep. DMA Technical Report 83502.2., Washington, DC, 1987.

[4] Samuel R Buss. *3D computer graphics: a mathematical introduction with OpenGL*. Cambridge University Press, 2003.

[5] D Salvio Carrijo, A Prieto Oliva, and W De Castro Leite Filho. "Hardware-in-loop simulation development". In: *International Journal of Modelling and Simulation* 22.3 (2002), pp. 167–175.

[6] cppreference. URL: http://en.cppreference.com/w/cpp/numeric/random/normal_distribution.

[7] Analog Devices. *ADIS16488, Tactical Grade Ten Degrees of Freedom Inertial Sensor*. [Online; accessed 25-May-2017]. 2014. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/ADIS16488.pdf.

[8] Olav Egeland and Jan Tommy Gravdahl. *Modeling and simulation for automatic control*. Vol. 76. Marine Cybernetics Trondheim, Norway, 2002.

[9] Eigen. URL: http://eigen.tuxfamily.org/index.php?title=Main_Page.

[10] Gazebo. URL: http://gazebosim.org/.

[11] Fathi Ghorbel, B Srinivasan, and Mark W Spong. "On the positive definiteness and uniform boundedness of the inertia matrix of robot manipulators". In: *Decision and Control, 1993., Proceedings of the 32nd IEEE Conference on*. IEEE. 1993, pp. 1103–1108.

[12] Martin Hepperle. *Propulsion by propellers*. [Online; accessed 25-May-2017]. 2005. URL: http://www.mh-aerotools.de/airfoils/prop_precession_english.htm.

[13] R Isermann, J Schaffnit, and S Sinsel. "Hardware-in-the-loop simulation for the design and testing of engine-control systems". In: *Control Engineering Practice* 7.5 (1999), pp. 643–653.

[14] Andreas Torp Karlsen. *On Modeling of a Ship Propulsion System for Control Purposes*. 2012.

[15] Henry Martin, Paul Groves, and Mark Newman. "The Limits of In-Run Calibration of MEMS Inertial Sensors and Sensor Arrays". In: *Navigation* 63.2 (2016), pp. 127–143.

[16] Microsoft. *AirSim*. URL: https://github.com/Microsoft/AirSim.

[17] Octomap. URL: https://octomap.github.io.

[18] OpenCV. *Camera Calibration and 3D Reconstruction*. [Online; accessed 20-May-2017]. 2017. URL: http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#fisheye.

[19] OpenSceneGraph. URL: http://www.openscenegraph.org.

[20] Deep Parikh, Jignesh Patel, and Jayesh Barve. "Quad-copter UAV BLDC Motor Control: Linear v/s non-linear control maps". In: *Nirma University Journal of Engineering and Technology* 4.1 (2015), p. 25.

[21] Minha Park and Yang Gao. "Error and performance analysis of MEMS-based inertial sensors with a low-cost GPS receiver". In: *Sensors* 8.4 (2008), pp. 2240–2261.

[22] S Parkes et al. "LIDAR-based GNC for Planetary Landing: Simulation with PANGU". In: *DASIA 2003*. Vol. 532. 2003.

[23] Alex G Quinchia et al. "A comparison between different error modeling of MEMS applied to GPS/INS integrated systems". In: *Sensors* 13.8 (2013), pp. 9549–9588.

[24] MA Shelley. "Monocular visual inertial odometry on a mobile device". PhD thesis. Master's thesis, Technischen Universitat Munchen, 2014.

[25] Joan Sola. *Quaternion Kinematics for the error-state Kalman filter*. 2017.

[26] Henning Thielemann. "Sampling-rate-aware noise generation". In: *arXiv preprint arXiv:1103.4118* (2011).

[27] Daniel Torres. "Sensorless BLDC control with back-EMF filtering using a majority function". In: *Microchip Technology* (2008).

[28] David A Vallado. *Fundamentals of astrodynamics and applications*. Vol. 12. Springer Science & Business Media, 2001.

[29] OpenGL Wiki. *Framebuffer Object — OpenGL Wiki,* [Online; accessed 25-May-2017]. 2016. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Framebuffer_Object&oldid=13801.

[30] OpenGL Wiki. *Language bindings — OpenGL Wiki,* [Online; accessed 19-May-2017]. 2017. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Language_bindings&oldid=13839.

[31] Wikipedia. *Bilinear interpolation — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Bilinear_interpolation&oldid=774133139.

[32] Wikipedia. *Gauss–Newton algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Gauss%E2%80%93Newton_algorithm&oldid=777220376.

[33] Wikipedia. *Homogeneous coordinates — Wikipedia, The Free Encyclopedia*. [Online; accessed 19-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Homogeneous_coordinates&oldid=775375830.

[34] Wikipedia. *Inertial frame of reference — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Inertial_frame_of_reference&oldid=766679372.

[35] Wikipedia. *Pinhole camera model — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Pinhole_camera_model&oldid=765888275.

[36] Wikipedia. *Propeller — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Propeller&oldid=777005778.

[37] Wikipedia. *Proper acceleration — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-June-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Proper_acceleration&oldid=764365523.

[38] Wikipedia. *Semi-implicit Euler method — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-June-2017]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Semi-implicit_Euler_method&oldid=744922107.

[39] Padmaraja Yedamale. "Brushless DC (BLDC) motor fundamentals". In: *Microchip Technology Inc* 20 (2003), pp. 3–15.